

Reabertura de Defeitos Corrigidos: Impactos e Prevenção

(Exame de Qualificação)

Rodrigo Rocha Gomes e Souza^{1,4}

Orientadora: Dra. Christina von Flach Garcia Chavez^{1,3}

Co-orientador: Dr. Roberto Almeida Bittencourt²

¹ Departamento de Ciência da Computação
Universidade Federal da Bahia (DCC-UFBA)

² Departamento de Ciências Exatas
Universidade Estadual de Feira de Santana (DEXA-UEFS)

³ Fraunhofer Project Center on Software and Systems Engineering
Universidade Federal da Bahia (FPC-UFBA)

⁴ Centro de Processamento de Dados
Universidade Federal da Bahia (CPD-UFBA)

1. Introdução

Corrigir defeitos é uma tarefa importante no desenvolvimento de software, muitas vezes apoiada por sistemas de acompanhamento de defeitos [Patton 2005]. Nesses sistemas são registradas, em relatórios de defeitos, informações sobre os defeitos encontrados no software. Em um cenário típico, abre-se um relatório para um defeito, o defeito é corrigido e, por fim, o relatório é fechado. Às vezes, no entanto, descobre-se que a correção foi incompleta ou inadequada, e nesse caso é preciso reabrir o relatório. Informalmente, diz-se que ocorreu a *reabertura do defeito*.

A reabertura de defeitos é pouco frequente, porém prejudicial ao processo de desenvolvimento de software. No projeto GNOME, por exemplo, cerca de 2% dos defeitos são reabertos [Almossawi 2012]. Em relação a defeitos resolvidos na primeira tentativa, no entanto, defeitos reabertos demoram duas vezes mais para serem corrigidos [Shihab et al. 2010] e envolvem a participação de 20 a 60% mais desenvolvedores [Park et al. 2012].

A reabertura de defeitos provoca retrabalho entre os desenvolvedores, causando frustração e diminuindo a produtividade da equipe [Almossawi 2012]. Além disso, a reabertura pode diminuir a qualidade do software como percebida por seus usuários e até mesmo atrasar o lançamento de novas versões.

Apesar de sua importância, apenas recentemente pesquisadores têm buscado entender o fenômeno da reabertura de defeitos [Shihab et al. 2010]. Já se constatou, por exemplo, que defeitos descobertos por usuários [Zimmermann et al. 2012] e defeitos em componentes mais complexos são mais propensos à reabertura [Almossawi 2012]. Ainda existem, no entanto, muitas questões a se investigar: qual o custo da reabertura no desenvolvimento de software? Que boas práticas podem ser adotadas para se prevenir a reabertura?

Muitas das pesquisas atuais ajudam a entender e até prever reabertura de defeitos¹. Nem sempre, no entanto, os resultados podem ser usados diretamente na prevenção da reabertura, uma vez que as previsões se baseiam em fatores que não podem ser controlados (como o componente onde o defeito se encontra [Shihab et al. 2010]).

Outra limitação dos estudos atuais se refere ao escopo estudado. Alguns relatórios de defeitos, por conterem informação insuficiente, não conseguem ser reproduzidos pelos desenvolvedores, que acabam por fechá-los durante a triagem de defeitos. Se o relatório é então atualizado e o desenvolvedor consegue reproduzir o defeito, ele é reaberto. Nestes casos, a reabertura ocorre por problemas no relato ou na triagem de defeitos. A maioria dos estudos não faz distinção entre essas reaberturas e as reaberturas que ocorrem por problemas na correção dos defeitos [Jongyindee et al. 2011] e acabam estudando os dois casos como um único fenômeno.

Por fim, ainda se sabe pouco sobre a magnitude dos impactos da reabertura de defeitos sobre o desenvolvimento de software. Os estudos atuais se limitam a relatar a taxa de reabertura (razão entre defeitos reabertos e defeitos relatados) [Almossawi 2012, Park et al. 2012], sem investigar, por exemplo, quanto tempo se gasta para tratar defeitos reabertos.

Neste trabalho pretendemos abordar as limitações do estado da arte. Sabe-se que um meio de evitar a reabertura de defeitos consiste em checar, por meio de técnicas de verificação de software (por exemplo, testes automatizados e revisão de código), se as correções são adequadas. Neste trabalho investigaremos como o processo de verificação pode ser otimizado para reduzir a ocorrência de defeitos reabertos.

Além disso, estamos interessados na reabertura de defeitos para os quais foi tentada uma correção, por considerarmos que este é um problema mais grave, que resulta em maior retrabalho. Daremos ênfase a defeitos que foram reabertos depois de terem sido corrigidos e verificados, uma vez que tais defeitos podem indicar oportunidades de melhoria do processo de verificação.

Por fim, pretendemos quantificar os impactos de defeitos reabertos. Mediremos não apenas a taxa de reabertura, mas também o custo das reaberturas com relação ao tempo adicional de desenvolvimento.

Uma visão mais abrangente do tema de pesquisa, com conceitos e trabalhos relacionados, encontra-se na Seção 2.

1.1. Objetivos

O objetivo geral deste trabalho é avaliar que características do processo de correção de defeitos (e posterior verificação dessas correções) contribuem para evitar a reabertura de defeitos corrigidos ou diminuir suas consequências negativas. A vantagem de estudar características dos processos é que o conhecimento obtido pode ser aplicado diretamente na sua melhoria. Espera-se, assim, determinar um conjunto de boas práticas de correção e verificação que auxiliem desenvolvedores a lidar com o problema da reabertura de defeitos.

Não será estudada a reabertura de defeitos por causa de problemas de triagem. Considera-se que este é um problema relacionado, porém diferente em relação a suas

¹Para um sumário do estado da arte, leia a Seção 2.7.

causas e à magnitude dos impactos negativos.

Com base na revisão da literatura, o objetivo geral é detalhado nos objetivos específicos listados a seguir:

1. *Caracterizar o processo de verificação a partir de artefatos gerados durante o desenvolvimento.* Este objetivo envolve entender as variáveis do processo de verificação de correções e encontrar meios de extrair tais variáveis a partir de sistemas de acompanhamento de defeitos.
2. *Caracterizar a reabertura de defeitos em termos de sua ocorrência e o tempo que ela consome durante o desenvolvimento.* Este objetivo envolve caracterizar a reabertura em termos de taxa de reabertura, severidade dos defeitos reabertos, tempo para correção, dentre outras variáveis.
3. *Inferir a influência do processo de verificação no custo de reaberturas.* Este objetivo envolve relacionar características do processo de verificação à ocorrência de reaberturas e ao tempo para correção. Serão investigadas também variáveis ligadas à correção do defeito, como o momento em que a correção é aplicada e desenvolvedor responsável pela correção. Variáveis como severidade dos defeitos, componente onde os defeitos se encontram, dentre outras, serão levadas em consideração.

Os objetivos são detalhados em questões de pesquisa na Seção 3.

1.2. Resultados Parciais

O primeiro objetivo específico, caracterizar o processo de verificação, já foi alcançado. Os resultados foram publicados na conferência MSR 2012 (9th Working Conference on Mining Software Repositories) [Souza and Chavez 2012]. Foi possível identificar, a partir de dados de sistemas de acompanhamento de defeitos de dois projetos de software livre — Eclipse e NetBeans —, as seguintes variáveis no processo de verificação:

- existência de uma fase de verificação bem definida precedendo o lançamento de uma nova versão do software vs. verificações bem distribuídas ao longo do ciclo de desenvolvimento;
- presença de uma equipe dedicada ao processo de verificação vs. desenvolvedores generalistas, que alternam entre programar e verificar;
- diferentes técnicas de verificação, como testes automatizados, testes *ad hoc* e revisão de código.

Durante o estudo, foi possível identificar e filtrar ruídos nos dados. Os ruídos incluem edições em massa dos relatórios de defeito, que não devem ser encaradas como registros das atividades dos desenvolvedores, e sim como uma limpeza dos dados.

Os demais objetivos específicos já foram abordados de maneira preliminar durante a pesquisa, como uma forma de avaliar a sua viabilidade. Um cronograma para a execução desses objetivos encontra-se na Seção 4.

Os resultados parciais da pesquisa são detalhados na Seção 5.

1.3. Contribuições

Até o momento, este trabalho resultou nas seguintes contribuições:

1. desenvolvimento de uma técnica para detectar ruído nos dados sobre verificação em sistemas de acompanhamento de defeitos (ver Seção 5.2.4);
2. desenvolvimento de técnicas para identificar, a partir de dados de sistemas de acompanhamento de defeitos, variáveis do processo de verificação de software: existência de uma equipe de qualidade, ocorrência de uma fase de verificação e uso de técnicas de verificação específicas (ver Seção 5.2.4);
3. avaliação empírica da eficácia do princípio dos quatro olhos como forma de evitar a reabertura de defeitos (ver Seção 5.3.1);

Espera-se, ainda, alcançar as seguintes contribuições:

1. construção de um mapeamento sistemático do estado da arte sobre reabertura de defeitos;
2. reprodução parcial do estudo de [Shihab et al. 2010], com foco em defeitos reabertos após correção e posterior comparação dos resultados;
3. desenvolvimento de teoria e técnica para computar o tempo de desenvolvimento adicional que pode ser atribuído a defeitos reabertos;
4. caracterização do tempo de desenvolvimento adicional que pode ser atribuído a defeitos reabertos, usando dados de alguns projetos, a fim de aumentar a compreensão sobre os custos da reabertura;
5. caracterização da reabertura de defeitos corrigidos em alguns projetos, a fim de aumentar a compreensão sobre o fenômeno;
6. avaliação empírica sobre a influência das variáveis do processo de verificação sobre a reabertura de defeitos.

Espera-se contribuir para a realidade do desenvolvimento de software através da elaboração de uma lista de boas práticas que, segundo os resultados empíricos obtidos na pesquisa, contribuem para evitar a reabertura de defeitos.

1.4. Organização

Este texto está organizado em 6 seções. Na Seção 2 é apresentada a revisão da literatura relevante sobre qualidade de software, reabertura de defeitos e temas relacionados. A seguir, na Seção 3, a proposta de pesquisa é detalhada. Na Seção 4 é apresentado o cronograma de atividades de pesquisa. Na Seção 5 são descritos os resultados obtidos até o momento e, por fim, na Seção 6 são apresentadas as considerações finais do trabalho.

2. Revisão de Literatura

A seguir são apresentados conceitos e estudos sobre temas relevantes a esta pesquisa. As primeiras seções tratam de conceitos como qualidade, verificação, acompanhamento de defeitos e métricas de qualidade. As seções finais tratam de trabalhos relacionados, incluindo estudos sobre reabertura de defeitos e mineração de repositórios de software.

2.1. Qualidade de Software

Quando se fala em qualidade e software, dois temas emergem: qualidade de produto e qualidade de processo. Acredita-se que, controlando a qualidade do processo de desenvolvimento de software, pode-se aprimorar a qualidade do produto de software desenvolvido.

2.1.1. Qualidade de Produto

No contexto de produtos de software, o termo *qualidade* se refere a atributos positivos, desejáveis, de sistemas de software. Naturalmente, a noção de qualidade é subjetiva, de modo que indivíduos diferentes podem ter percepções de qualidade distintas sobre um mesmo produto. Tal subjetividade não impede que, ao longo do tempo, diversos indivíduos e organizações tenham buscado sistematizar o conceito de qualidade de software, dando origem a *modelos de qualidade*.

Modelos de qualidade de software definem um conjunto de características a serem consideradas para se avaliar a qualidade de um software [ISO/IEC 2001]. Exemplos de modelos de qualidade incluem o modelo de Boehm [Boehm et al. 1976], o modelo de Cavano e McCall [Cavano and McCall 1978], o modelo FURPS+ [Grady 1992] e a ISO/IEC 9126 [ISO/IEC 2001].

No modelo de Cavano e McCall, as características de qualidade são agrupadas em três dimensões:

- *revisão do produto*: manutenibilidade, flexibilidade e testabilidade;
- *transição do produto*: portabilidade, reusabilidade e interoperabilidade;
- *operação do produto*: corretude, confiabilidade, eficiência, integridade e usabilidade.

Ainda que antigo, o modelo de Cavano e McCall propõe uma classificação ainda hoje relevante — basta observar que o padrão ISO 9126 usa o modelo como uma das fontes de inspiração. Além disso, cada dimensão se refere a atributos de qualidade que são especialmente relevantes para um stakeholder: a dimensão de revisão é relevante para desenvolvedores; transição, para administradores de sistemas; operação, para o usuário.

Neste trabalho, são enfocados os atributos de qualidade referentes à operação do produto. Estes são os atributos que afetam diretamente os usuários.

2.1.2. Anomalias de Software

Erro, falta, falha, defeito, *bug*, problema, irregularidade... Muitos termos são usados quando o software não se comporta como o esperado durante a sua operação, resultando na diminuição da qualidade percebida pelo usuário.

A fim de reduzir ambiguidades, o padrão IEEE 1044-2009 define uma terminologia para anomalias de software, composta dos termos defeito, erro, falha, falta e problema.

- *falha*: evento no qual o software não desempenha a função requerida;
- *defeito*: imperfeição no software que precisa ser reparada para que o software passe a atender seus requisitos ou especificação;
- *falta*: tipo de defeito encontrado durante a execução do software (provocando, portanto, uma falha);
- *erro*: uma ação humana que produz um resultado incorreto, por exemplo, um erro de codificação;
- *problema*: dificuldade ou incerteza vivenciada por uma ou mais pessoas, resultante de um encontro insatisfatório com um sistema em uso, por exemplo, impossibilidade de se autenticar em um sistema.

Para ilustrar o uso dos termos, eis uma situação hipotética: “Maria relata que ela não consegue se autenticar no sistema porque o campo de senha não está aparecendo na tela de autenticação” (adaptado do IEEE 1044). No exemplo, Maria se deparou com um *problema* (a impossibilidade de se autenticar no sistema), causado por uma *falha* (o não-aparecimento do campo de senha), por sua vez causado por um *erro* de programação que introduziu um *defeito* no código-fonte.

Neste trabalho, utilizamos o termo *defeito* em um sentido amplo, análogo ao sentido comumente atribuído termo inglês *bug*, que engloba as imperfeições no software propriamente ditas, suas consequências em um sistema em execução e o seu relatório em um sistema de acompanhamento de defeitos. Assim, para manter uma consistência com termos usados na literatura, e sem prejuízo para o entendimento, são usadas expressões como *relatório de defeito* — ainda que, na maioria dos casos, os relatórios descrevam falhas — e *reabertura de defeito* — ainda que a reabertura se refira ao relatório, e não ao defeito.

2.1.3. Qualidade de Processo

Quando se fala em qualidade de software, pode-se estar referindo também a características desejáveis do processo de desenvolvimento. A motivação para controlar a qualidade do processo é a expectativa de produzir software de qualidade de maneira controlada.

Dentre os modelos de qualidade de processo de software, provavelmente o mais conhecido é o *Capability Maturity Model Integration* (CMMI) [Chrissis et al. 2003], do *Software Engineering Institute* (SEI). Ele define 5 níveis de maturidade para organizações que desenvolvem software, segundo características do processo de desenvolvimento:

- nível 1, inicial: processos com pouco controle;
- nível 2, gerenciado: processos definidos individualmente para cada projeto; áreas de verificação e validação fazem parte do nível de maturidade 3, juntamente com outras 9 áreas de processo.
- nível 3, definido: processo definido para a organização;
- nível 4, quantitativamente gerenciado: o processo é avaliado e controlado quantitativamente;
- nível 5, em otimização: o processo é continuamente aprimorado.

Na versão 1.3 do CMMI, são definidas ainda 22 áreas de processo, que descrevem aspectos do processo de desenvolvimento que deveriam ser cobertos pelas organizações. Por exemplo, as áreas de verificação e validação fazem parte do nível de maturidade 3, juntamente com outras 9 áreas de processo.

Alguns pesquisadores têm defendido que a comunidade deve se concentrar em práticas reusáveis, destinadas a resolver aspectos específicos do desenvolvimento de software, em detrimento de processos completos [Jacobson et al. 2007]. Seguindo essa filosofia, este trabalho se propõe a buscar práticas relacionadas à correção de defeitos e à verificação de software que lidem com o problema da reabertura de defeitos.

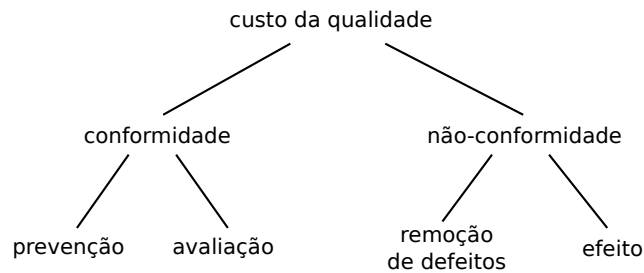


Figura 1. Custos associados à qualidade de software. Adaptado de [Wagner 2005].

2.1.4. O Custo da Qualidade

O ideal de equipes de desenvolvimento de software é entregar software com alta qualidade em pouco tempo e a baixo custo. As pressões de tempo e custo por vezes levam equipes a dar menos prioridade a atividades voltadas à melhoria de qualidade, como testes e inspeções de código, acreditando que assim estão economizando tempo e reduzindo custos. A relação entre qualidade e custo, no entanto, não é tão direta.

Os diferentes custos associados à qualidade estão esquematizados na Figura 1. Existem dois tipos gerais de custos: os *custos de conformidade* e os *custos de não-conformidade* [Slaughter et al. 1998].

Os custos de conformidade se referem às atividades realizadas para produzir software de qualidade. Eles podem ser divididos em *custos de prevenção* e *custos de avaliação*. Os custos de prevenção incluem treinamentos, reuniões e a aquisição de ferramentas destinadas a prevenir a produção de software de má qualidade. Já os custos de avaliação englobam testes, revisão de código, medição de software e qualquer prática voltada para a avaliação da qualidade.

Os custos de não-conformidade englobam os gastos associados a software de baixa qualidade. Estão aí incluídos os *custos de remoção de defeitos* e os *custos de efeito*. Os custos de remoção de defeitos, frequentemente associados a retrabalho, ocorrem quando um defeito é descoberto. Quando isso ocorre, surgem custos que incluem a localização do defeito no código-fonte (que pode envolver a depuração do software), desenvolvimento de uma correção e verificação da correção. Os custos de efeito [Wagner 2006] são todos os demais custos que surgem quando o software falha diante de usuários. Estes podem incluir, dentre outros, o custo da diminuição da base de usuários e o custo de compensação financeira de usuários que sofreram danos causados por falhas de software.

Por um lado, iniciativas voltadas à melhoria de qualidade envolvem custos adicionais, os custos de conformidade. Por outro lado, não desenvolver tais iniciativas pode levar a custos de não-conformidade, causados pela baixa qualidade do software.

2.2. Verificação e Validação de Software

Verificação é o processo de checar se o software está de acordo com os requisitos e atributos de qualidade definidos de maneira implícita ou explícita pela equipe. Validação é o processo de checar se o software cumpre seus propósitos, isto é, se atende às necessidades de seus usuários. Informalmente, diz-se que validação é checar se você “construiu

o produto certo”, e verificação é checar se você “construiu certo (i.e., corretamente) o produto” [IEEE Computer Society 2004].

Frequentemente verificação e validação são tratados como um único tópico, abreviado por V&V [IEEE Computer Society 2004]. Neste trabalho, não é feita a distinção entre os dois conceitos; optamos por chamar de verificação qualquer checagem de qualidade, seja em relação aos requisitos ou em relação à satisfação do usuário.

As técnicas de verificação de software são classificadas em dinâmica ou estática [Sommerville 2006]. Nas técnicas dinâmicas, também conhecidas como teste, o software é executado e seu comportamento é comparado com o comportamento esperado a fim de se detectar defeitos. Já nas técnicas estáticas o software é analisado sem ser executado. As técnicas estáticas podem ser usadas não somente para identificar defeitos, como também para checar se determinadas convenções de codificação e boas práticas estão sendo seguidas, se o código viola decisões arquiteturais etc.

As técnicas de verificação podem ser executadas automaticamente ou manualmente. Como os testes consistem em seguir roteiros e comparar resultados obtidos com resultados esperados, muitos testes podem ser automatizados. Ainda assim, é comum que durante o desenvolvimento sejam realizados testes *ad hoc*, isto é, testes manuais realizados de forma não sistemática.

A verificação estática também pode ser realizada em parte por ferramentas automáticas. É o caso de ferramentas de análise estática de código [Rutar et al. 2004], capazes de detectar defeitos comuns como aqueles relacionados a alocação de memória, inicialização de variáveis ou concorrência, ou ainda más práticas e variações no estilo de codificação. Já a revisão de código é uma forma de verificação estática que consiste na leitura do código por um ou mais desenvolvedores, os quais buscam irregularidades.

Dentro de um projeto, o processo de verificação pode ser conduzido pelos próprios desenvolvedores ou por uma equipe separada. Neste último caso, diz-se que o projeto adota verificação e validação independente (IV&V, do inglês *independent verification and validation*) [Callahan et al. 1998, Arthur et al. 1999], e que existe uma equipe de QA (garantia de qualidade, do inglês *quality assurance*). As atribuições de uma equipe de QA vão além da verificação, e podem incluir a criação e aplicação de padrões e métodos para melhorar o processo de desenvolvimento e evitar o surgimento de defeitos, monitorar o processo, decidir quando a próxima versão do software está pronta para ser lançada, entre outras.

2.3. Acompanhamento de Defeitos

Sistemas de acompanhamento de defeitos permitem que usuários e desenvolvedores de um sistema gerenciem uma lista de defeitos do projeto, juntamente com informações como severidade do defeito, passos para reproduzir a falha ou o sistema operacional utilizado durante a execução do software. Para cada defeito são registradas também discussões e o progresso de resolução do defeito, incluindo sua correção e verificação. Apesar do nome, sistemas de acompanhamento de defeitos são comumente usados para registrar também novas funcionalidades desejadas [Antoniol et al. 2008].

Cada registro em um sistema de acompanhamento de defeitos é chamado de relatório de defeito ou tíquete, embora tíquete seja um termo mais neutro, aplicável também

a requisições de funcionalidades. Outro termo empregado é requisição de mudança.

Embora cada sistema de acompanhamento de defeitos tenha suas peculiaridades, eles tendem a ser similares quanto às informações básicas registradas no ato da criação de um tíquete e durante o progresso da resolução de defeitos. A título de ilustração, serão descritas as informações registradas pelo sistema Bugzilla.

O Bugzilla foi lançado em 1998², sendo um dos primeiros sistemas de acompanhamento de defeitos open source com interface web. Ele é até hoje um sistema popular, usado em projetos como Eclipse, Mozilla, Linux Kernel, NetBeans, Apache e companhias como NASA e Facebook³.

Ao criar um tíquete no Bugzilla, o usuário se depara com uma página como a que é mostrada na Figura 2. Nessa página ele especifica qual o produto e o componente que apresentaram defeito, bem como a versão do produto onde o defeito foi encontrado. Além disso, ele pode informar o ambiente de execução do software (*hardware* e sistema operacional), bem como a severidade do defeito (ex.: crítico, trivial). Por fim, deve escrever um sumário do defeito, e opcionalmente uma descrição mais longa. A descrição idealmente deve conter informações como os passos que devem ser executados para que o problema se manifeste, o comportamento obtido e o esperado, e quaisquer mensagens de erro exibidas pelo software.

O sistema registra os dados do usuário que criou o tíquete, a data de abertura, e atribui ao tíquete um número identificador. A partir daí, usuários podem adicionar comentários a fim de esclarecer o problema e as circunstâncias em que ele ocorre, discutir soluções, solicitar colaboração de desenvolvedores etc. Também é possível anexar arquivos (por exemplo, imagens, logs, patches). Outra possibilidade é alterar as informações registradas ou acrescentar novas informações.

A Figura 3 mostra a página do tíquete #72317 do Eclipse. No início da página são mostrados o número e o sumário do tíquete. A seguir, são exibidos os dados informados durante a criação do tíquete e mais alguns dados que podem ser atualizados pelos desenvolvedores, como o *target milestone* (versão-alvo, i.e., versão do sistema na qual se espera que o defeito esteja resolvido), *assigned to* (desenvolvedor responsável pelo tíquete) e o *status* (i.e., o progresso da resolução do tíquete). É possível também especificar relacionamentos entre tíquetes, marcando um tíquete como duplicata de outro, ou registrando que a resolução de um tíquete depende da resolução de outro. No final, é listada a descrição do tíquete e todos os comentários que sucederam sua criação.

Outro dado que os desenvolvedores podem atualizar é a prioridade do tíquete, i.e., com que urgência o defeito deve ser corrigido. A prioridade pode assumir valores de P1, mais prioritário, até P5, menos prioritário. Na Figura 3, a prioridade é exibida juntamente com a severidade no campo *importance* (no exemplo, a prioridade é P3 e a severidade é normal). Alguns projetos relacionam prioridade a lançamento de versões. Por exemplo, em um determinado subprojeto do Eclipse⁴, uma nova versão só pode ser lançada depois que todos os tíquetes P1 forem resolvidos. Tíquetes P2 também têm alta prioridade, mas não chegam a impedir o lançamento se não forem resolvidos a tempo.

²<http://www.bugzilla.org/status/roadmap.html>

³Uma lista mais completa pode ser encontrada em <http://www.bugzilla.org/installation-list/>

⁴<http://www.eclipse.org/tpth/home/documents/process/development/bugzilla.html>

Before reporting a bug, please read the [bug writing guidelines](#), please look at the list of [most frequently reported bugs](#), and please [search](#) for the bug.

Show Advanced Fields

(* = Required Field)

*** Product:** Platform

*** Component:** Ant
Compare
CVS
Debug
Doc
IDE
Incubator

*** Version:** 4.0
4.1
4.2
4.2.1
4.3

Reporter: rodrigorgs+eclipse@gmail.com

Component Description:
Select a component to read its description.

Severity: normal

Hardware: PC

OS: Linux

We've made a guess at your operating system and platform. Please check them and make any corrections if necessary.

*** Summary:**

Description:

Attachment:

Figura 2. Página de relato de um novo defeito no Bugzilla do Eclipse.

Já a severidade de um defeito está relacionada aos impactos negativos que ele pode causar para seus usuários. Os possíveis valores podem ser personalizados pelo projeto, mas os seguintes valores são mais comuns⁵:

- *blocker*: atrapalha o desenvolvimento ou os testes;
- *critical*: causa perda de dados, encerramento do programa etc.;
- *major*: perda significativa de funcionalidade;
- *normal*: alguma perda de funcionalidade sob condições específicas;
- *minor*: problema no qual há uma solução paliativa;
- *trivial*: problemas cosméticos, como erros de ortografia;
- *enhancement*: não é um defeito, e sim um pedido de melhoria.

O valor de qualquer atributo de um defeito pode ser alterado a qualquer momento. Assim, por exemplo, um defeito inicialmente marcado com severidade *normal* pode ser posteriormente marcado como *critical* se surgirem novas informações que indicam que o defeito provoca perda de dados.

⁵https://developer.mozilla.org/en-US/docs/What_to_do_and_what_not_to_do_in_Bugzilla



BUGS

[CONTACT](#) | [LEGAL](#)





Bugzilla – Bug 72317 Copying from IOConsole inserts extra line breaks. Last modified: 2004-09-13 10:34:48 EDT

[Home](#) | [New](#) | [Browse](#) | [Search](#) | [\[2\]](#) | [Reports](#) | [Requests](#)
[Help](#) | [Log In](#) | [Forgot Password](#) | [Terms of Use](#) | [Copyright Agent](#)

First Last Prev Next This bug is not in your last search results.

Bug 72317 - Copying from IOConsole inserts extra line breaks.

Status: VERIFIED FIXED

Reported: 2004-08-19 17:42 EDT by Kevin Barnes

Product: Platform

Modified: 2004-09-13 10:34 EDT ([History](#))

Component: Debug

CC List: 3 users ([show](#))

Version: 3.0

See Also:

Platform: All All

Importance: P3 normal ([vote](#))

Target Milestone: 3.1 M2

Assigned To: Darin Swanson

QA Contact:

URL:

Whiteboard:

Keywords:

Depends on:

Blocks:

Show dependency [tree](#)

Attachments

[Add an attachment](#) (proposed patch, testcase, etc.)

Note

You need to [log in](#) before you can comment on or make changes to this bug.

Kevin Barnes 2004-08-19 17:42:24 EDT

[Description](#)

when pasting multiple lines into IOConsole, they get double spaced.

Kevin Barnes 2004-08-27 23:36:38 EDT

[Comment 1](#)

This got fixed along with the ProcessConsole work.

Kevin Barnes 2004-08-27 23:37:02 EDT

[Comment 2](#)

Darin, please verify.

Figura 3. Tíquete #72317 do Eclipse.

- **DUPLICATE**: o mesmo defeito já foi relatado em outro tíquete;
- **WONTFIX**: o defeito não será corrigido;
- **WORKSFORME**: o desenvolvedor não conseguiu reproduzir o defeito;
- **INVALID**: o problema descrito não é um defeito;

No Bugzilla do Eclipse, o status e a resolução são ambos exibidos no campo *status*, como mostra a Figura 3. No exemplo, o tíquete foi corrigido (resolução **FIXED**) e verificado (status **VERIFIED**).

Naturalmente, cada projeto tem a liberdade de usar o Bugzilla de acordo com suas necessidades, de forma adaptada a seu processo. Por exemplo, uma equipe pequena, que não realize verificação de código novo, pode não usar o *status VERIFIED* ou, ainda, usá-lo para outros fins, como para sinalizar que a correção já está disponível no repositório de código central do projeto.

2.4. Métricas de Qualidade

Como já foi dito na Seção 2.1.1, qualidade é um atributo desejado porém subjetivo. Por esta razão é necessário definir métricas para capturar diversos aspectos da qualidade de software. As métricas de qualidade de software dividem-se em dois grupos gerais [Kan 2002]:

- métricas de produto: descrevem atributos de qualidade intrínsecas ao produto;
- métricas de processo: descrevem a eficiência e eficácia do processo de desenvolvimento em relação à melhoria de qualidade do produto;

A seguir são descritas algumas métricas de qualidade encontradas na literatura [Kan 2002]: uma métrica de produto — densidade de defeitos — e duas métricas de processo — tempo para correção e correções defeituosas.

2.4.1. Densidade de Defeitos

O principal indicador da qualidade de um produto é a sua quantidade de defeitos: quanto mais defeitos, pior a qualidade. No entanto, o número absoluto não é um bom indicador, pois espera-se que, quanto maior o sistema, maior seja o seu número de defeitos. Por isso é importante considerar os tamanhos dos sistemas e o seu crescimento ao longo do tempo.

A métrica *densidade de defeitos* [Kan 2002] é uma medida da quantidade de defeitos de um sistema de software em relação ao seu tamanho. Para medir o tamanho de um sistema podem ser empregadas diversas métricas, como quantidade de pontos de função, número de unidades ou número de linhas de código (LOC). A métrica LOC é mais fácil de se calcular e, por isso, é empregada com maior frequência. Para evitar lidar com valores fracionários muito pequenos, é comum considerar milhares de linhas de código (KLOC).

$$\text{densidade de defeitos} = \frac{\text{número de defeitos}}{\text{KLOC}}$$

Uma desvantagem de se usar LOC para calcular a densidade de defeitos é que programas idênticos escritos em linguagens de programação distintas podem apresentar

tamanhos em LOC muito diferentes. A métrica é adequada, no entanto, para comparar diferentes versões de um mesmo sistema.

Outra questão é que existem diversas formas de se medir o número de linhas de código de um sistema. Algumas pessoas contam todas as linhas de cada arquivo fonte, outros ignoram linhas em branco e comentários, entre outras variações. É importante definir claramente como se pretende medir LOC e se ater à definição.

2.4.2. Tempo para Correção

Um defeito que demora para ser corrigido contribui para a diminuição da satisfação dos usuários e pode atrasar o lançamento de novas versões do software. A métrica *tempo para correção* [Kan 2002] (do inglês *time-to-fix*) mede o tempo que leva desde o relato até a correção de um defeito. Como indicador de qualidade do projeto como um todo, calcula-se o tempo médio para correção — ou, se os tempos forem muito heterogêneos, a mediana. É conveniente realizar o cálculo separadamente para cada severidade ou prioridade que os defeitos podem assumir.

$$\text{tempo para correção (de um defeito)} = \text{instante da correção} - \text{instante da abertura}$$

$$\text{tempo médio para correção} = \frac{\sum (\text{instante da correção} - \text{instante da abertura})}{\text{número de defeitos corrigidos}}$$

2.4.3. Correções Defeituosas

Não é suficiente que a correção para um defeito seja encontrada a tempo: a correção deve, ela própria, ser livre de defeitos. Uma correção é defeituosa quando não corrige completamente o defeito ao qual se destina ou quando introduz novos defeitos. Em outras palavras, é uma correção que resulta na reabertura do defeito no futuro.

Uma métrica simples relacionada a correções defeituosas é a *taxa de reabertura* [Kan 2002]:

$$\text{taxa de reabertura} = \frac{\text{número de defeitos reabertos após correção}}{\text{número de defeitos corrigidos}}$$

Alguns autores acreditam que a taxa de reabertura pode ser um indicador otimista para sistemas grandes, com grande número de defeitos corrigidos: nestes, mesmo uma porcentagem pequena pode representar um elevado número absoluto de reaberturas [Kan 2002]. Uma alternativa é considerar o tamanho do sistema:

$$\text{reaberturas} = \frac{\text{número de defeitos reabertos após correção}}{\text{KLOC}}$$

Pode-se registrar dois instantes de uma correção defeituosa: o instante em que a correção foi proposta e o instante em que o defeito foi reaberto. O intervalo de tempo entre os dois instantes corresponde ao *período latente da correção defeituosa* [Kan 2002]:

$$\text{período latente} = \text{instante da reabertura} - \text{instante da correção}$$

Longos períodos latentes são perigosos porque implicam longo tempo para correção: um defeito não pode ser corrigido enquanto não se sabe que ele ainda se manifesta. Ainda pior, se o período latente englobar a data do lançamento de uma versão, o produto pode ser lançado com um defeito que se acreditava estar corrigido.

2.5. Mineração de Repositórios de Software

Repositórios de software são coleções de artefatos que são produzidos e arquivados durante a evolução de um sistema de software [Kagdi et al. 2007]. Alguns exemplos de tipos de repositório de software [Hassan 2008]:

- *Sistemas de controle de versão.* Arquivam múltiplas versões de arquivos textuais, sobretudo código-fonte, mas podem incluir também arquivos de configuração e documentação. Cada versão está associada a metadados como data em que a alteração foi realizada, nome do desenvolvedor que realizou a alteração, e uma mensagem descrevendo as mudanças. Exemplos de sistemas de controle de versão incluem o CVS, o Subversion e o Git.
- *Sistemas de acompanhamento de defeitos* (ou *repositórios de defeitos*). Arquivam relatórios de defeitos e solicitações de mudança. Tais repositórios ajudam os desenvolvedores a registrar, discutir e acompanhar a situação de cada solicitação, desde a atribuição de uma solicitação a um desenvolvedor até a verificação da correção ou melhoria. Exemplos incluem o Bugzilla, o Trac e o Jira.
- *Listas de discussão.* Representam um canal de comunicação entre as pessoas envolvidas em um projeto, na qual são discutidos os mais diversos aspectos do projeto. Podem ser usadas também para notificar participantes de um projeto sobre eventos do sistema de controle de versão e do sistema de acompanhamento de defeitos.
- *Outros repositórios.* Outras fontes de dados sobre um projeto podem incluir o seu site, fóruns de discussão e canais de bate-papo (IRC).

A mineração de repositórios de software tem como objetivo aproveitar o registro histórico contido nos repositórios para extrair informações úteis para o desenvolvimento futuro de um sistema em particular [Hassan 2008] ou mesmo para validar hipóteses sobre o desenvolvimento de software em geral.

O acesso a repositórios de software é um aspecto crítico da pesquisa na área, e tradicionalmente se dava por meio de cooperações com grandes empresas. Atualmente, com a popularização do movimento de software livre e de código aberto, qualquer pesquisador pode ter acesso a repositórios de projetos de grande porte, desenvolvidos por grandes equipes e com uma vasta base de usuários.

A disponibilidade de repositórios fez com que a mineração de repositórios de software tenha se popularizado como instrumento de pesquisa em engenharia de software. A conferência Mining Software Repositories, dedicada ao tema, é realizada desde 2004.

A seguir são apresentadas algumas pesquisas que têm como base a mineração de repositórios de defeitos.

Predição de defeitos. As técnicas de predição de defeitos envolvem a análise do código-fonte de uma versão de um sistema [Basili et al. 1996, Marcus et al. 2008] ou de dados históricos do controle de versão [Moser et al. 2008, Nagappan et al. 2010] e de sistemas de acompanhamento de defeitos [Kim et al. 2007]. A finalidade é prever a ocorrência ou o número de defeitos que serão encontrados em cada componente do sistema (pacote, classe, método ou qualquer outra unidade de código) [D’Ambros et al. 2010].

Mapeamento de defeitos para código-fonte. Para validar algoritmos de predição de defeitos é necessário ter um sistema de referência no qual se saiba de antemão o número de defeitos por componente. Para descobrir essa informação é necessário mapear cada defeito registrado no sistema de acompanhamento de defeitos para os componentes de código que apresentam o defeito. O estado da arte do mapeamento automático consiste buscar *commits* que corrigem um defeito específico relatado no sistema de acompanhamento de defeitos (referenciado na mensagem de *commit* pelo seu número identificador). Considera-se então que os componentes alterados no *commit* correspondem a componentes que apresentavam o defeito referenciado [Fischer et al. 2003, Śliwerski et al. 2005, Eaddy et al. 2008, D’Ambros et al. 2010].

Viés no mapeamento de defeitos para código-fonte. A eficácia das técnicas automáticas de mapeamento de defeitos para código depende da disciplina dos programadores em referenciar defeitos nas mensagens de *commit*. Na prática, apenas uma fração dos defeitos consegue ser mapeada [Ayari et al. 2007]. O problema é que essa fração é uma amostra enviesada dos defeitos [Bird et al. 2009, Nguyen et al. 2010]. Por exemplo, a amostra de defeitos mapeados contém uma proporção maior de defeitos severos e de defeitos verificados do que o conjunto de todos os defeitos. Esse viés compromete a eficácia e a análise de algoritmos de predição de defeitos e qualquer outra análise que dependa do mapeamento. Há três sugestões para mitigar o problema [Bird et al. 2009]: buscar conjuntos de dados mais completos e confiáveis, mapear os defeitos manualmente, e treinar modelos em uma amostra dos dados cuja distribuição de características como severidade, experiência etc. seja próxima à da população.

Perigos na mineração de sistemas de acompanhamento de defeitos. Nem toda a coordenação realizada entre desenvolvedores a fim de corrigir um defeito está registrada precisamente no sistema de acompanhamento de defeitos; o desenvolvedor ao qual o defeito está atribuído nem sempre participa da resolução do defeito [Aranda and Venolia 2009]. Por exemplo, pode acontecer de o defeito ter sido resolvido antes mesmo da criação de seu relatório, ou de o desenvolvedor só lembrar de marcar o defeito como resolvido muito tempo depois de sua resolução. Deve-se tomar cuidado também com ruído nos dados e edições em massa dos relatórios [Aranda and Venolia 2009]. Outro problema é que nem todos os tíquetes marcados como defeitos correspondem a manutenção corretiva; alguns correspondem a manutenção perfectiva ou preventiva, discussões sobre arquitetura etc. [Antoniol et al. 2008, Herzig et al. 2012].

Nas próximas seções são apresentados outros trabalhos, mais diretamente relacionados a este, que empregam mineração de repositórios de software.

2.6. Tempo de Vida de Defeitos

Uma informação relevante que pode ser extraída de sistemas de acompanhamento de defeitos é o tempo de correção de cada defeito. Tempos de correção longos podem indicar componentes do sistema mais difíceis de se manter [Kim and Whitehead 2006] e até mesmo atrasar o lançamento de novas versões.

Diversas abordagens têm sido aplicadas para prever o tempo de correção de um defeito. Uma das abordagens envolve o emprego de algoritmos de mineração de dados [Panjer 2007] e de regressão [Anbalagan and Vouk 2009] aplicados a atributos de tíquetes. Outra abordagem consiste em usar métricas de similaridade para prever o tempo de correção de um tíquete com base nos tempos de tíquetes semelhantes já corrigidos, seja explorando a similaridade textual entre os títulos e descrições dos tíquetes [Weiss et al. 2007] ou agrupando tíquetes com atributos semelhantes (severidade, tipo de problema etc.) através de uma rede neural [Zeng and Rine 2004].

A seguir são listadas algumas conclusões de estudos sobre o tempo de vida de defeitos:

- defeitos com maior severidade são corrigidos mais rapidamente [Panjer 2007, Hooimeijer and Weimer 2007, Bougie et al. 2010, Zhang et al. 2012];
- tíquetes cuja descrição é mais fácil de ler são corrigidos mais rapidamente [Hooimeijer and Weimer 2007];
- tíquetes com muitos comentários demoram mais para ser corrigidos [Panjer 2007, Hooimeijer and Weimer 2007, Duc Anh et al. 2011];
- quanto maior o número de participantes em um tíquete, maior o tempo de correção [Anbalagan and Vouk 2009, Duc Anh et al. 2011];
- o desempenho passado de um desenvolvedor é um bom preditor do tempo que ele levará para corrigir um defeito [Duc Anh et al. 2011];
- defeitos de vida longa são caracterizados por certas construções de código [Canfora et al. 2011];
- defeitos que demoram mais para ser atribuídos a um desenvolvedor são corrigidos mais rapidamente após a atribuição [Ohira et al. 2012];
- quando o defeito é reportado por uma pessoa, triado por outra e corrigido por uma terceira, o tempo de correção é duas vezes maior [Ohira et al. 2012];
- outros atributos que ajudam a prever o tempo de vida de um defeito incluem a sua categoria [Bougie et al. 2010], o mês em que o defeito foi aberto, o desenvolvedor atribuído para corrigir o defeito [Giger et al. 2010] e a pessoa que relatou o defeito [Giger et al. 2010] (embora existam divergências sobre a importância deste último atributo [Bhattacharya and Neamtiu 2011]).

2.7. Reabertura de Defeitos

Depois que um defeito é resolvido, pode-se descobrir que a resolução não foi adequada e então é necessário reabrir o defeito antes de realizar uma nova resolução. A reabertura de defeitos pode indicar que o software é instável ou difícil de manter [Baker and Eick 1994] e representa um custo adicional no tempo de desenvolvimento, uma vez que equivale a um esforço que não culminou na resolução definitiva do problema. Além disso, a reabertura pode trazer consigo diversas consequências negativas:

- *Diminuição da qualidade percebida pelo usuário.* Defeitos reabertos que não são corrigidos a tempo acabam se manifestando no próximo lançamento do software, afetando a experiência do usuário e diminuindo a sua confiança no sistema.
- *Retrabalho por parte dos desenvolvedores.* Os desenvolvedores precisam investir tempo novamente na correção de um defeito que já deveria ter sido corrigido. O retrabalho envolve tanto a correção em si quanto a repetição de atividades de verificação, a depender do projeto.
- *Frustração da equipe.* Acertar na primeira tentativa é bom; retrabalho causa frustração, uma vez que o tempo poderia ser melhor empregado corrigindo outros defeitos ou implementando novas funcionalidades.
- *Custo adicional.* Naturalmente, retrabalho significa tempo e custo adicionais. Além disso, deve-se considerar que o custo de se corrigir um defeito é menor quanto mais cedo ele é encontrado, uma vez que outras mudanças no software podem ter impacto sobre o código defeituoso.
- *Atraso na data de lançamento.* Quando as funcionalidades previstas para a próxima versão são implementadas e os defeitos mais sérios são corrigidos, é lançada uma nova versão do software. Se um defeito for reaberto tardiamente no ciclo de lançamentos, a data de lançamento pode ser adiada, frustrando as expectativas dos usuários e dos desenvolvedores.

Um defeito pode ser resolvido de diversas formas, e isso é geralmente indicado no sistema de acompanhamento de defeitos. No Bugzilla, por exemplo, existe um campo do relatório de defeitos, “resolução”, que pode assumir valores como `WORKSFORME`, `INVALID`, `DUPLICATE`, `WONTFIX` e `FIXED` (ver Seção 2.3.1).

Quando o desenvolvedor não consegue reproduzir a falha descrita em um relatório, o defeito é resolvido como `WORKSFORME`. Depois disso, um dos participantes do tíquete pode fornecer informações adicionais. Se as informações forem suficientes para se reproduzir o defeito, o desenvolvedor então reabre o tíquete.

Situação similar ocorre quando o defeito é marcado como `INVALID` (não é um defeito), `DUPLICATE` (existe outro tíquete para o mesmo defeito) ou `WONTFIX` (é um defeito cuja correção é inviável ou cujos impactos negativos são negligenciáveis). Informações adicionais podem revelar que o comportamento exibido pelo programa realmente é anômalo, que é diferente do que foi relatado em outros tíquetes, ou que o defeito é mais severo do que se pensava, e também nesses casos o tíquete é reaberto.

Nos casos já mencionados, existe um esforço de triagem e classificação de defeitos que precisa ser feito quando surgem novas informações. Frequentemente o problema é falta de informação no relatório de defeito, e os desenvolvedores não podem agir até que sejam fornecidas mais informações.

Um problema mais sério ocorre quando um defeito é reconhecido como válido, recebe uma correção (i.e., é resolvido como `FIXED`), e então se descobre que a correção não foi adequada, seja porque não cobriu todos os casos ou porque introduziu novos defeitos. Nesse caso os desenvolvedores realizaram uma triagem do tíquete, classificaram-no, localizaram o defeito no código-fonte e corrigiram-no. Ao descobrir que o defeito ainda se manifesta em determinados casos especiais, é necessário localizar os casos especiais no código fonte, possivelmente usando depuração, e desenvolver uma nova solução, mais adequada. Esse nível de retrabalho apresenta um custo mais elevado e causa frustração na

equipe, que criou a expectativa de que o defeito havia sido corrigido.

Neste trabalho estamos interessados neste último caso: a reabertura de defeitos considerados corrigidos. Ainda que qualquer tipo de reabertura traga consigo prejuízos, no caso específico de defeitos supostamente corrigidos o custo de reabertura é maior. Além disso, existe uma oportunidade de evitar esse tipo de reabertura através da melhoria do processo de verificação de correções — por exemplo, revisando o código das correções no fim do ciclo de lançamento de versões ou instituindo uma equipe dedicada ao processo de verificação.

A seguir serão apresentados alguns dados e conclusões presentes na literatura sobre reabertura de defeitos. Nota-se que muitos trabalhos estudam reabertura como um fenômeno único, coerente, sem diferenciar entre os tipos de resolução que precederam a reabertura.

2.7.1. Por Que os Defeitos são Reabertos?

Zimmermann e colegas [Zimmermann et al. 2012] conduziram uma pesquisa na Microsoft Research perguntando a seus funcionários as razões pelas quais um defeito seria reabertos múltiplas vezes antes de ser corrigido. As respostas levaram às seguintes causas comuns:

- os desenvolvedores não conseguiram reproduzir o defeito e o fecharam como `WORKSFORME` (ou o equivalente no sistema de defeitos usado pela Microsoft);
- a causa raiz do defeito não foi corretamente identificada e por isso os desenvolvedores acabaram corrigindo um defeito diferente;
- um defeito inicialmente marcado como `WONTFIX` teve sua prioridade aumentada depois que surgiram novas informações;
- o desenvolvedor deixou passar um caso especial em sua correção que foi descoberto durante os testes;
- a equipe de testes avaliou a correção como adequada, mas depois se descobriu que a correção não era completa.
- a equipe de testes avaliou uma versão do software no qual a correção ainda não havia sido aplicada, e por isso reabriu o tíquete;

As causas levantadas se referem a defeitos cuja resolução final foi `FIXED`, embora as resoluções iniciais que levaram à reabertura sejam diversas. As causas mais frequentemente mencionadas, segundo os autores, estão ligadas à dificuldade de se reproduzir os defeitos.

Almossawi [Almossawi 2012] estudou apenas a reabertura de defeitos considerados corrigidos em 32 sistemas integrantes do projeto GNOME entre a versão 2.0 e a versão 2.32, de 2002 a 2010, totalizando 951 defeitos. A partir de uma amostra de 210 desses defeitos, o autor identificou as seguintes causas:

- em 68% dos casos, o tíquete foi reaberto porque o defeito reapareceu;
- em 16% dos casos, as causas estavam ligadas a erros humanos ou falhas na colaboração, como esquecer de submeter o *patch*, interpretar erroneamente a correção ou não compreender o componente corrigido devido a falta de documentação;
- em 7% dos casos, o defeito regrediu, isto é, ele se manifestou por causa de outros defeitos ou modificações realizadas em outros componentes;

- os demais defeitos foram reabertos por causas diversas, não citadas pelo autor.

Nota-se que a causa mais frequente de reabertura de defeitos marcados como `FIXED` são correções inadequadas, que são o foco deste trabalho. É esse tipo de reabertura que pode ser evitado através da melhoria do processo de correção de defeitos.

Park e colegas [Park et al. 2012] levantaram erros de programação que levam à necessidade de se submeter mais de uma correção para o mesmo problema. Eles analisaram uma mostra 100 defeitos que foram referenciados em mais de um *commit* no sistema de controle de versões dos projetos Eclipse e Mozilla. Em alguns casos, isso representa mais de uma tentativa de corrigir o defeito, configurando a reabertura do defeito. Em outros casos, no entanto, isso apenas quer dizer que o desenvolvedor optou por dividir sua tentativa de correção em mais de um *commit* a fim de tornar sua contribuição mais clara para possíveis revisores. Essa diferenciação não é feita no trabalho. De qualquer forma, foram levantadas as seguintes categorias de erros associados à ocorrência de múltiplos *commits* por defeito:

- a primeira mudança não cobria todas as plataformas nas quais o produto executa;
- havia uma instrução condicional incorreta no primeiro *commit*;
- determinada unidade de código foi modificada, mas faltou modificar unidades associadas a ela;
- a correção inicial propunha a criação de uma API que foi posteriormente julgada inadequada;
- a correção incluía uma refatoração incompleta;

2.7.2. Em que Defeitos Reabertos Diferem dos Demais?

Identificar as causas da reabertura de defeitos é importante para evitar os erros que levam a esse fenômeno. Não menos importante é descobrir em que aspectos os defeitos reabertos diferem dos demais; essa descoberta pode levar a modelos para identificar defeitos e correções propensos a reabertura, de forma a concentrar esforços de verificação nesses defeitos.

Shihab e colegas [Shihab et al. 2010, Shihab et al. 2012] desenvolveram um modelo de árvore de decisão para prever a reabertura de defeitos do Eclipse, versão 3.0, com base em dados disponíveis no sistema de acompanhamento de defeitos e no sistema de controle de versões. O modelo apresentou 62,9% de precisão e 84,5% de revocação. Não foi feita distinção entre as diferentes resoluções que levaram às reaberturas. Foram considerados 22 fatores que podem influenciar a reabertura de defeitos, incluindo características do relatório de defeito, características da correção, além de fatores humanos. Os seguintes fatores foram considerados mais importantes na predição de reaberturas:

- descrição do defeito e comentários escritos no relatório do defeito: certos termos, como “depuração”, “breakpoint” e “plataformas” estão associadas a defeitos reabertos;
- tempo que demorou para o defeito receber a primeira correção: quanto mais tempo demora, maior a chance de reabertura;
- componente no qual o defeito foi encontrado: certos componentes de um sistema são mais problemáticos do que outros.

Naturalmente, esses fatores não podem ser controlados dentro de um projeto. Nenhuma equipe vai deixar de tentar corrigir defeitos em certos componentes porque sabe que a chance de reabertura é alta. O que pode ser feito, no entanto, é direcionar os esforços de verificação para esses componentes.

Zimmermann e colegas [Zimmermann et al. 2012] desenvolveram um modelo de regressão logística para explicar a reabertura de defeitos no Windows Vista e no Windows 7. Eles chegaram às seguintes conclusões:

- defeitos encontrados através de revisão de código ou de ferramentas de análise de código têm menos chance de serem reabertos, supostamente por serem mais fáceis de triar e resolver;
- por outro lado, defeitos encontrados por usuários ou durante o teste de sistema são mais propensos a reabertura, por serem mais complexos e difíceis de se reproduzir;
- defeitos inicialmente atribuídos a uma pessoa que não pertence à mesma equipe de quem abriu o relatório de defeito têm maior chance de serem reabertos;
- defeitos com maior severidade inicial têm maior chance de serem reabertos;

Jongyindee e colegas [Jongyindee et al. 2011] investigaram defeitos reabertos após receberem uma correção no Eclipse. Eles descobriram que defeitos corrigidos por desenvolvedores que contribuem mais ativamente com código-fonte têm menor chance de serem reabertos.

Almossawi [Almossawi 2012] concluiu que defeitos reabertos após serem corrigidos tendem a ser caracterizados por código-fonte com alta complexidade ciclomática, e que defeitos em geral estão associados a código com alto acoplamento. O resultado sugere que a reabertura de defeitos é caracterizados por um tipo de complexidade diferente do que favorece a ocorrência de defeitos em geral.

Caglayan e colegas [Caglayan et al. 2012] estudaram um sistema corporativo a fim de identificar fatores que caracterizam a reabertura de defeitos. Ao treinar modelos de regressão logística sobre os dados do sistema de acompanhamento de defeitos, eles determinaram que os seguintes fatores são importantes para a reabertura:

- atividade dos desenvolvedores: são mais propensos a reabertura os defeitos atribuídos a desenvolvedores que corrigiram defeitos mais recentemente ou que editam muitos métodos no código-fonte;
- centralidade do defeito: são mais propensos a reabertura os defeitos em métodos associados a outros defeitos;
- localização geográfica: são mais propensos a reabertura defeitos atribuídos a pessoas geograficamente afastadas de quem criou o relatório de defeito.

Além disso, foram entrevistados desenvolvedores da equipe de qualidade da empresa para aumentar a compreensão sobre os resultados encontrados. Os desenvolvedores consideraram que alguns dos resultados podem ser explicados por dois fatores que propiciam a reabertura: *complexidade do defeito* e *carga de trabalho dos desenvolvedores*.

2.7.3. Qual o Custo de Defeitos Reabertos?

Duas métricas têm sido utilizadas na literatura para caracterizar o custo da reabertura: a taxa de reabertura e o tamanho do ciclo de vida de defeitos reabertos. Os trabalhos dife-

rem, no entanto, com relação a quais reaberturas são consideradas: alguns só consideram reaberturas que sucedem correções, outros computam qualquer tipo de reabertura.

Taxa de reabertura. Considerando apenas defeitos corrigidos, a taxa de reabertura é de 2,15% no projeto GNOME [Almossawi 2012], 1,35% no Evolution [Almossawi 2012], 3,88% no GTK+ [Almossawi 2012], e varia entre 9,0% e 9,5% no Eclipse e no NetBeans [Wang et al. 2011]. Analisando defeitos que podem ser vinculados a *commits* no repositório de código-fonte, conclui-se que 11,73% dos *commits* de correção no Eclipse resultam em reabertura [Jongyindee et al. 2011].

Ciclo de vida. Defeitos reabertos apresentam um ciclo de vida mais longo, isto é, o período desde a sua abertura até sua resolução final e fechamento é maior do que o de defeitos que foram resolvidos em uma tentativa. Para Shihab e colegas [Shihab et al. 2010], o ciclo de vida de defeitos reabertos (independentemente de terem sido corrigidos) é duas vezes mais longo. Para Jongyindee e colegas [Jongyindee et al. 2011], defeitos reabertos após terem sido corrigidos e verificados têm um ciclo de vida cerca de 5 vezes maior do que os demais defeitos válidos (isto é, defeitos que não foram marcados como inválidos ou duplicados).

Jongyindee e colegas [Jongyindee et al. 2011] identificaram seis tipos de reabertura e alertaram que nem todas apresentam impacto negativo sobre o projeto. Eles notam que algumas vezes defeitos são propositalmente fechados para serem resolvidos em algum momento no futuro, quando as condições forem mais propícias. A reabertura, nesse caso, é planejada. A recomendação é que mais pesquisadores prestem atenção ao tipo de reabertura em seus trabalhos.

3. Proposta de Pesquisa e Métodos

Como enunciado na Seção 1.1, o objetivo geral deste trabalho é avaliar que características do processo de correção de defeitos e verificação contribuem para evitar a reabertura de defeitos corrigidos ou diminuir suas consequências negativas. A seguir são delineados os métodos gerais de pesquisa. Então são enunciados os objetivos específicos, juntamente com questões de pesquisa e métodos associados.

3.1. Métodos

Como método geral de pesquisa, emprega-se neste trabalho a mineração de repositórios de software (ver Seção 2.5), usando como fontes de dados os sistemas de acompanhamento de defeitos de alguns projetos selecionados para estudo de caso. São realizadas análises qualitativas e quantitativas sobre os dados.

Primeiramente é realizada uma análise exploratória dos dados. A análise envolve a criação de gráficos a partir dos dados, bem como a leitura de documentação e de amostras de relatórios de defeitos de cada projeto. Essa primeira etapa tem como objetivo conhecer melhor os projetos e os dados disponíveis, identificar necessidades de limpeza e filtragem dos dados, bem como levantar hipóteses e variáveis de interesse.

A seguir, são construídos programas para classificar os relatórios de defeitos segundo as variáveis levantadas anteriormente. Por exemplo, um programa pode ser usado para determinar, com base nos relatórios de defeitos, se existe uma equipe de qualidade (QA) e, em caso afirmativo, quais defeitos foram verificados por essa equipe. Nesse caso,

os defeitos são separados em duas classes — QA e não-QA —, segundo o papel do desenvolvedor responsável por sua verificação.

Após a classificação ocorre uma segunda rodada de análise exploratória, desta vez considerando a classificação dos relatórios de defeitos. São construídos novos gráficos, segmentados por classe, e são lidos relatórios de defeitos a partir de uma amostra aleatória de cada classe. O objetivo é testar e refinar o programa de classificação e, ao mesmo tempo, compreender melhor a variável sendo estudada.

Estes são os métodos gerais usados na pesquisa. Os métodos específicos para cada objetivo e cada questão de pesquisa são detalhados a seguir.

3.2. Objetivo: Caracterizar o Processo de Verificação

O primeiro objetivo específico é caracterizar o processo de verificação dos projetos selecionados e a forma como o processo é registrado em sistemas de acompanhamento de defeitos. O objetivo é alcançado por meio da investigação das seguintes questões de pesquisa, de caráter exploratório:

- RQ1.1: Quando a verificação é realizada dentro do ciclo de vida de lançamentos do projeto? Ela é realizada logo após cada correção ou apenas no final do ciclo, caracterizando uma fase de verificação?
- RQ1.2: Quem realiza a verificação? Existe uma equipe de qualidade especializada em verificações ou todos os desenvolvedores realizam verificações?
- RQ1.3: Que técnicas são empregadas para a verificação de correções? Inspeção de código, testes automatizados, testes *ad hoc*?
- RQ1.4: Que ameaças existem ao se investigar as questões RQ1.1, RQ1.2 e RQ1.3 usando dados de sistemas de acompanhamento de defeitos?

O método de pesquisa envolve três etapas: extração, amostragem e análise de dados. A extração é a obtenção de dados disponíveis publicamente de sistemas de acompanhamento de projetos de software livre. A amostragem, no caso de projetos grandes, divididos em subprojetos, consiste em escolher um número reduzido de subprojetos representativos do projeto maior. A análise da amostra é específica para cada questão de pesquisa:

- RQ1.1: Quando a verificação é realizada? Selecionam-se, para cada subprojeto, todas as verificações relatadas (isto é, mudança de *status* para VERIFIED) durante toda a vida do projeto. Então plota-se, para cada dia no intervalo, o número acumulado de verificações até aquele dia. Além disso, marcam-se nos gráficos as datas de lançamento de novas versões dos sistemas. Analisando o gráfico é possível identificar se, próximo às datas de lançamento, há uma intensificação da atividade de verificação.
- RQ1.2: Quem realiza a verificação? A fim de determinar se existe uma equipe de qualidade dedicada à verificação, contam-se quantas vezes cada desenvolvedor marcou defeitos como FIXED ou VERIFIED. Considera-se que um desenvolvedor é parte de uma equipe de qualidade se ele realizou, digamos, 10 vezes mais verificações do que correções⁶. Ao final deste processo, considera-se qual proporção das verificações foi realizada pela equipe de qualidade computada. Se essa

⁶O número 10 é arbitrário; pode-se avaliar a sua adequação analisando-se a distribuição da razão entre verificações e correções por desenvolvedor.

porcentagem for baixa, considera-se que não existe, de fato uma equipe de qualidade dedicada a verificações.

- RQ1.3: Que técnicas são empregadas para a verificação de correções? São analisados os comentários que os desenvolvedores escrevem quando marcam um defeito como `VERIFIED` ou `REOPENED`. Buscam-se então certas palavras ou expressões chave que se referem a técnicas de verificação (ex.: “testes de unidade”, “leitura do código”, ou suas traduções para o inglês). Se essas expressões forem encontradas, considera-se que a técnica correspondente foi empregada. Ao final, conta-se a frequência com que cada técnica é empregada.
- RQ1.4: Ameaças relacionadas às questões anteriores. Aproveitam-se os gráficos da RQ1.1 e buscam-se padrões anômalos de crescimento da atividade de verificação ao longo do tempo. Sendo encontrados tais padrões, parte-se para a leitura dos relatórios de defeitos contidos nos períodos anômalos para se entender melhor a ameaça.

3.3. Objetivo: Caracterizar a Reabertura de Defeitos

O segundo objetivo consiste em caracterizar a reabertura de defeitos em termos da frequência com que ela ocorre e do prejuízo de tempo que ela traz ao projeto. Apesar de existirem trabalhos semelhantes na literatura (ver Seção 2.7), esta proposta se diferencia por dar ênfase a dois subconjuntos de defeitos reabertos: os defeitos que foram corrigidos e os defeitos que foram corrigidos e verificados. O objetivo é alcançado por meio da investigação das seguintes questões de pesquisa, de caráter exploratório:

- RQ2.1: Como a reabertura de defeitos se distribui em relação ao tempo, em relação aos desenvolvedores e em relação a características de defeitos? Esta questão de pesquisa envolve a aferição das seguintes métricas:
 - taxa de reabertura (Seção 2.4.3) de todos os defeitos;
 - taxa de reabertura dos defeitos que são corrigidos;
 - taxa de reabertura dos defeitos que são corrigidos e depois verificados;
 - distribuição da quantidade de reaberturas por resolução (ex.: `FIXED`, `VERIFIED`, `WONTFIX`);
 - distribuição da quantidade de reaberturas ao longo do ciclo de lançamento;
 - distribuição da quantidade de reaberturas por desenvolvedor que corrigiu ou verificou o defeito;
 - distribuição da quantidade de reaberturas por papel na equipe do desenvolvedor que corrigiu ou verificou o defeito (programador ou especialista em qualidade);
 - distribuição da quantidade de reaberturas por papel do desenvolvedor no defeito reaberto (o papel, neste caso, indica se o desenvolvedor foi quem relatou ou quem corrigiu o defeito);
 - relação entre a reabertura e o tempo para correção (Seção 2.4.3);
- RQ2.2: Qual o impacto dos defeitos reabertos em termos do esforço de desenvolvimento, em relação aos defeitos não reabertos? São investigadas as seguintes métricas:
 - tempo para correção (Seção 2.4.3) dos defeitos que não foram reabertos;
 - tempo para verificação dos defeitos corrigidos que não foram reabertos;
 - tempo para correção final dos defeitos corrigidos e reabertos;

- tempo para verificação final para os defeitos corrigidos e reabertos;
- tempo para correção dos defeitos corrigidos reabertos e não reabertos, verificados e não verificados, por severidade do defeito;
- período latente de correções defeituosas (tempo entre correção e reabertura; ver Seção 2.4.3);
- período latente dos defeitos corrigidos e verificados;
- relação entre o tamanho do período latente e o tempo para correção subsequente (será que defeitos que demoram mais para ser reabertos demoram mais para ser recorrigidos?);
- dentre os defeitos reabertos, o tempo para correção dos defeitos corrigidos;
- dentre os defeitos reabertos, o tempo para correção dos defeitos não corrigidos (ou seja, com outra resolução);

A primeira etapa para alcançar o objetivo é realizar uma revisão da bibliografia relevante, em busca de respostas para essas questões e questões relacionadas. Um sumário da bibliografia sobre reabertura de defeitos é apresentado na Seção 2.7. A revisão tem sido atualizada à medida que surgem novos artigos sobre o tema.

Na análise dos dados de sistemas de acompanhamento de defeitos, as métricas poderão ser computadas separadamente para defeitos de diferentes severidades, componentes etc., a fim de isolar o efeito dessas características nas métricas. Considera-se, ainda, realizar uma replicação parcial de algum dos estudos de predição e caracterização de reaberturas [Shihab et al. 2010, Zimmermann et al. 2012].

3.4. Objetivo: Investigar a Influência do Processo de Correção e Verificação na Reabertura de Defeitos

Cumpridos os objetivos específicos apresentados anteriormente, adquire-se uma compreensão abrangente do fenômeno de reabertura de defeitos e suas consequências negativas, bem como do processo de verificação de correções e suas nuances. Cabe então investigar a relação entre as duas coisas: até que ponto a verificação de correções reduz a ocorrência e os impactos da reabertura de defeitos? Este objetivo envolve a investigação das seguintes questões de pesquisa:

- RQ3.1: Qual o impacto do instante da verificação na reabertura de defeitos? Mais especificamente:
 - é mais eficaz (para evitar reaberturas) ter uma fase de verificação ou verificar defeitos o tempo todo?
 - dentre os defeitos reabertos, quais demoram mais para serem corrigidos⁷: aqueles verificados em uma fase de verificação ou fora dela?
- RQ3.2: Qual o impacto da equipe de qualidade na reabertura de defeitos? Mais especificamente:
 - defeitos verificados pela equipe de qualidade apresentam menos chance de ser reabertos?
 - defeitos verificados pela equipe de qualidade, quando reabertos, apresentam um tempo de vida menor?
- RQ3.3: Qual o impacto da técnica de verificação na reabertura de defeitos? Mais especificamente:

⁷Calcula-se o intervalo de tempo decorrido entre a reabertura e a correção subsequente.

- quais técnicas de verificação contribuem mais para evitar a reabertura de defeitos?
- quais técnicas de verificação contribuem para reduzir o tempo de vida de defeitos reabertos?

Além dos métodos empregados anteriormente, são empregados testes estatísticos de hipótese, mais especificamente testes de associação entre as variáveis, como o teste exato de Fisher. O planejamento detalhado dos estudos ainda está sendo realizado. Segue, a título de exemplo, um esboço dos passos que devem ser seguidos ao se estudar a relação entre equipe de qualidade e reabertura (RQ3.2):

- são selecionados todos os tíquetes de um projeto;
- tíquetes de requisição de melhorias são descartados;
- tíquetes que sofreram alguma edição em massa são descartados;
- são descartados todos os tíquetes cuja última atualização tenha ocorrido menos de um ano⁸ antes do último dia disponível nos dados; o objetivo é selecionar apenas tíquetes cujo histórico completo esteja disponível — considera-se que, se o tíquete passou um ano sem ser modificado, ele provavelmente não será mais modificado;
- são identificados os usuários que possuem mais de uma conta no sistema de acompanhamento de defeitos e são criados identificadores únicos para eles [Bird et al. 2006];
- reaberturas que ocorrem poucos dias (quantos?) depois da correção são desconsideradas, sob a justificativa de que seu impacto sobre o desenvolvimento é baixo;
- são descartadas as verificações relatadas pela pessoa que corrigiu o defeito, e que ocorrem alguns minutos (quantos?) depois da correção, por se considerar que, nesse caso, a correção e a verificação não foram executadas como tarefas bem separadas;
- os defeitos são classificados de acordo com sua severidade, sua prioridade, e do componente que apresentou defeito.
- identifica-se o papel de cada desenvolvedor, isto é, se ele faz parte da equipe de qualidade (QA) ou não (ver Seção 5.2.4).
- são selecionados apenas os tíquetes verificados;
- os tíquetes são classificados segundo o papel do verificador, i.e., do desenvolvedor que o verificou: QA ou não-QA;
- constrói-se uma tabela de contingência, com a contagem de tíquetes agrupados em duas dimensões: reaberto/não-reaberto e QA/não-QA.
- executa-se o teste exato de Fisher para determinar, com certo nível de confiança, se há associação entre o papel do verificador e a reabertura de defeitos;
- por fim, verifica-se se existe associação quando outras variáveis são levadas em consideração (prioridade, severidade, componente); o método estatístico a ser empregado ainda é um ponto de discussão.

O último ponto é destinado a isolar o efeito do papel do verificador sobre a reabertura. Pode-se descobrir, por exemplo, que a equipe de qualidade tende a verificar os defeitos mais severos (e possivelmente mais difíceis de corrigir e verificar), e por isso tende a provocar mais reaberturas, proporcionalmente, do que os demais desenvolvedores, que tendem a verificar defeitos mais fáceis. Diz-se, neste caso, que a severidade é uma

⁸Mais de 90% dos tíquetes dos projetos Eclipse/Platform e NetBeans/Platform são reabertos menos de um ano depois de serem corrigidos.

variável de confusão, papel do desenvolvedor é uma *variável de exposição* e reabertura é uma *variável de efeito*.

Existem diversas abordagens para controlar variáveis de confusão, de forma a se medir a influência da variável de exposição na variável de efeito (no exemplo, efeito do papel do desenvolvedor na reabertura de defeitos). Dois métodos comuns que estão sendo considerados nesta pesquisa são a estratificação e a regressão [McNamee 2005].

Na estratificação, a variável de confusão é dividida em estratos (isto é, subgrupos) e então se comparam os efeitos em cada estrato, de acordo com a exposição. No exemplo, pode-se calcular a taxa de reabertura para cada severidade e cada papel (exposição), e então computar a razão entre as duas taxas em cada nível de severidade (estrato). Por fim, pode-se obter uma razão geral ao calcular a média das razões, ponderada de acordo com a prevalência do estrato na população.

Na regressão, constrói-se um modelo que relaciona as variáveis explanatórias (variáveis de confusão e de exposição) à variável de efeito através de uma expressão matemática⁹. O coeficiente encontrado para a variável de exposição pode ser interpretado como a variação esperada na variável de efeito quando a variável de exposição aumenta em uma unidade, mantendo as demais variáveis constantes. Assume-se, no entanto, que as variáveis explanatórias não estão altamente relacionadas.

4. Cronograma

Por se tratar de uma pesquisa em andamento, alguns dos objetivos e questões de pesquisa apresentados na Seção 3 já foram abordados. Em particular, o primeiro objetivo específico (Seção 3.2, caracterizar o processo de verificação) foi cumprido [Souza and Chavez 2012].

Os demais objetivos já foram abordados de maneira preliminar, e por isso ainda precisam ser mais explorados. Um relato de todos os resultados de pesquisa já obtidos é apresentado na Seção 5.

4.1. Atividades

As seguintes atividades, relacionadas aos objetivos específicos não completados, estão previstas para a conclusão deste trabalho:

1. Caracterizar a reabertura de defeitos (segundo objetivo específico, Seção 3.3).
2. Escrever um artigo sobre a caracterização da reabertura de defeitos.
3. Investigar a influência do processo de correção e verificação na reabertura de defeitos (terceiro objetivo específico, Seção 3.4).
4. Escrever a tese.
5. Apresentar a tese.
6. Escrever um artigo com os resultados finais da tese.

A Tabela 1 relaciona os períodos previstos para a execução de cada atividade.

4.2. Plano de Publicações

Com base na data proposta, no tema e nos métodos do artigo previsto na atividade 2, foram identificadas algumas conferências para as quais o artigo pode ser submetido, as quais são listadas na Tabela 2.

⁹Um método mais elaborado, que usa regressão, é o *propensity score matching*, que leva em consideração a probabilidade de cada indivíduo ser submetido à variável de exposição.

Tabela 1. Cronograma de atividades. 2013.4 representa o quarto trimestre do ano de 2013.

Atividade	2012.4	2013.1	2013.2	2013.3	2013.4	2014.1
1	•	•				
2		•	•			
3			•	•		
4				•		
5					•	•
6						•

Tabela 2. Conferências relevantes para o artigo previsto na atividade 2

Conferência	Prazo	Qualis
ESEM 2013 - Empirical Software Engineering and Measurement	01/03/2013	A2
ESEC/FSE 2013 - Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering	01/03/2013	A2
ICSM 2013 - IEEE International Conference on Software Maintenance	14/04/2013	A2
PROFES 2013 - International Conference on Product Focused Software Development and Process Improvement	31/01/2013	B1
MSR 2013 - Working Conference on Mining Software Repositories	08/02/2013	B1
SBES 2013 - Simpósio Brasileiro de Engenharia de Software	?	B2
SBQS 2013 - Simpósio Brasileiro de Qualidade de Software	?	B3
PROMISE 2013 - International Conference on Predictor Models in Software Engineering	?	?

Com relação ao artigo previsto na atividade 6, este pode ser submetido a novas edições das conferências listadas, à conferência CSMR (*European Conference on Software Maintenance and Engineering*), ou a diversos periódicos, como *IEEE Transactions on Software Engineering*, *Empirical Software Engineering*, *Information and Software Technology*, *Software Practice and Experience*, *Journal of Systems and Software*, dentre outros.

5. Resultados Parciais

A seguir estão listados os resultados obtidos até o momento com esta pesquisa. O primeiro objetivo específico, caracterizar o processo de verificação, foi alcançado [Souza and Chavez 2012]. Os demais objetivos se encontram em diferentes graus de progresso. Os resultados já alcançados e os resultados preliminares são relatados a seguir.

Tabela 3. Sumário dos dados disponíveis sobre os projetos estudados

Projeto	# Subprojetos	# Relatórios	Primeiro Relatório	Último Relatório
Eclipse	155	316.911	outubro de 2001	junho de 2010
NetBeans	39	185.578	junho de 1998	junho de 2010

5.1. Projetos Estudados

Foram estudados até então dois projetos de software livre, Eclipse¹⁰ e NetBeans¹¹, ambos ambientes de desenvolvimento integrados voltados para Java, mas com *plugins* para diversas outras linguagens. Ambos usam o Bugzilla como sistema de acompanhamento de defeitos.

Os dados utilizados foram obtidos do *website* do MSR 2011 Challenge¹², parte do evento Mining Software Repositories. São disponibilizados *dumps* dos bancos de dados MySQL usados pelas instalações do Bugzilla do Eclipse e do Netbeans. Os arquivos contêm todos os dados dos respectivos bancos de dados, incluindo comentários e mudanças de *status*, exceto os perfis dos desenvolvedores, que foram omitidos por questões de privacidade. Um sumário dos dados encontra-se na Tabela 3.

O desenvolvimento do Eclipse começou no final de 1998 com a IBM¹³. Ele foi licenciado como software livre em novembro de 2001. Já o NetBeans começou como um projeto de estudantes¹⁴ em 1996. Ele então foi comprado pela Sun Microsystems em Outubro de 1999 e tornado software livre em março de 2000.

5.2. Caracterização dos Projetos

Os projetos, em particular seu processo de lançamento e verificação, foram investigados por meio de leitura da documentação do projeto e de análise qualitativa e quantitativa de seus relatórios de defeitos, como apresentado a seguir. A caracterização se mostrou importante para conhecer melhor os projetos e os dados disponíveis sobre eles.

5.2.1. Documentação

Ambos os projetos possuem documentação voltada para o desenvolvedor em seus respectivos *websites*. Foram procurados documentos relacionados ao acompanhamento de defeitos.

No projeto Eclipse, segundo uma apresentação voltada para a comunidade do projeto, uma nova versão *milestone* é lançada a cada seis semanas¹⁵. O projeto possui documentação específica sobre o uso do Bugzilla, na qual se encontra uma orientação sobre o processo de verificação: quando um defeito é corrigido, é responsabilidade de outro desenvolvedor no time verificar a correção¹⁶. Segundo o documento, é importante que sejam pessoas diferentes porque quem corrigiu está muito próximo ao código e pode não ser tão

¹⁰<http://eclipse.org/>

¹¹<http://netbeans.org/>

¹²<http://2011.msrrconf.org/msr-challenge.html>

¹³<http://www.ibm.com/developerworks/rational/library/nov05/cernosek/>

¹⁴<http://netbeans.org/about/history.html>

¹⁵<http://www.eclipsecon.org/2005/presentations/econ2005-eclipse-way.pdf>

¹⁶http://wiki.eclipse.org/Development_Resources/HOWTO/Bugzilla_Use

cuidadoso ao testar casos excepcionais. Essa recomendação é conhecida em alguns meios como princípio dos quatro olhos [Rubin et al. 2007].

Além disso, todos os defeitos devem ser verificados antes do próximo *build* de integração. Quando o projeto lança uma versão *major*, os defeitos verificados são fechados.

No projeto NetBeans, os desenvolvedores são classificados em desenvolvedores internos (*core*), desenvolvedores externos (*modules*) e engenheiros de qualidade¹⁷. A expectativa no projeto é que quem relata o defeito é responsável por verificar a sua resolução, uma vez que ele possui o conhecimento e as ferramentas necessárias para reproduzir o defeito, ou então por engenheiros de qualidade. Novas versões só podem ser lançadas após a resolução de defeitos de prioridade P1 (mais prioritários).

As informações obtidas na documentação dos projetos contribuem para o primeiro objetivo específico deste trabalho, caracterizar o processo de verificação (Seção 3.2).

5.2.2. Análise Qualitativa de Relatórios de Defeito

A fim de investigar a reabertura de defeitos e o processo de correção de defeitos, foram selecionados dois subprojetos do Eclipse — Platform e EMF — e dois subprojetos do NetBeans — VersionControl e Profiler. Os subprojetos foram escolhidos dentre os subprojetos com pelo menos 10 defeitos reabertos. A seguir, foram selecionados 20 defeitos de cada subprojeto. Cada relatório de defeito foi lido integralmente a fim de se conhecer melhor o *modus operandi* de cada projeto.

Defeitos foram classificados em duas dimensões: se o desenvolvedor que resolveu o defeito foi o mesmo que o relatou (sim ou não), e se o defeito foi reaberto (sim ou não), totalizando quatro configurações possíveis. Então, foram selecionados aleatoriamente 5 defeitos para cada configuração, para cada subprojeto, totalizando 80 defeitos. A seguir são apresentadas algumas conclusões.

No projeto Eclipse/EMF, o status `VERIFIED` não é usado para sinalizar a realização da verificação do defeito, e sim para sinalizar que a correção do defeito já foi aplicada ao código-fonte e está disponível em um *build* no *website* do projeto. Essa prática foi decidida em reunião cuja ata foi referenciada em um dos defeitos amostrados¹⁸.

Nos defeitos amostrados dos subprojetos do NetBeans, a situação mais frequente foi o defeito ser verificado pela mesma pessoa que o relatou, o que está de acordo com a recomendação do *website*. Em uma análise quantitativa posterior, percebeu-se que, em 80% a 90% dos casos, os relatores são os próprios engenheiros de qualidade [Souza and Chavez 2012].

No Eclipse/Platform, foi encontrado um padrão de colaboração entre desenvolvedores na resolução e verificação de defeitos. Frequentemente o desenvolvedor que corrige o defeito solicita a outro desenvolvedor que verifique a correção. Se este desenvolvedor decide que a correção não é adequada e reabre o defeito, ele propõe uma nova correção e solicita ao primeiro desenvolvedor que verifique a correção, e assim sucessivamente (como no tíquete número 14758). Esse padrão sugere que não há desenvolvedores especializados em qualidade; em vez disso, os desenvolvedores são generalistas.

¹⁷<http://wiki.netbeans.org/IssuesHandling>

¹⁸<http://wiki.eclipse.org/ModelingPMCMeeting,2007-10-16>

Em alguns casos, o defeito é reaberto porque foi marcado como `RESOLVED`, `VERIFIED` ou `CLOSED` por engano, ou porque a marcação foi incorretamente, por falta de conhecimento sobre as práticas do projeto. É o caso do tíquete número 77407 do NetBeans.

Tais descobertas ajudam a caracterizar a reabertura e portanto contribuem para o segundo objetivo específico deste trabalho (Seção 3.3).

5.2.3. Identificação de Causas de Reabertura

No contexto de uma disciplina da pós-graduação da qual fui monitor, foi solicitado que os alunos lessem relatórios de defeitos reabertos em 4 projetos de software livre — Adium, CakePHP, Joomla e Planner — e tentassem descobrir a causa da reabertura com base em informações disponíveis nos relatórios. Eis algumas situações descobertas pelos alunos:

- Projeto: Adium (<http://adium.im/>)
 - O defeito foi reaberto porque não cobria todos os casos.
 - O defeito foi reaberto porque a correção estava no nível de abstração errado (deveria ter sido aplicada em uma biblioteca, e não no código que usa a biblioteca).
- Projeto: Cake PHP (<http://cakephp.org/>)
 - O tíquete foi marcado como resolvido porque o desenvolvedor entendeu (incorretamente) que era uma duplicata.
 - O próprio contribuidor identificou que a correção foi incompleta.
 - O próprio contribuidor identificou que a correção gerou novos defeitos.
 - O tíquete foi marcado como resolvido porque o defeito havia sido resolvido em uma versão de desenvolvimento, mas o usuário precisava de uma resolução na última versão estável.
 - Um desenvolvedor marcou como resolvido porque achou que outro desenvolvedor havia resolvido, quando isso não ocorreu de fato.
 - A descrição do tíquete não estava detalhada o suficiente, e o desenvolvedor marcou como inválido por entender incorretamente.
- Projeto: Joomla (<http://www.joomla.org/>)
 - Solução incompleta.
 - Faltou criar um tíquete no sistema de acompanhamento de defeitos antes de marcar como resolvido.
- Projeto: Planner (<https://live.gnome.org/Planner>)
 - O tíquete foi marcado como resolvido e então reaberto porque a contribuição externa não havia sido revisada e nem aplicada ao repositório de código.

As causas descobertas são consistentes com as causas levantadas na literatura (veja a Seção 2.7.1). Encontram-se, nesta lista, reaberturas provocadas por correções inadequadas, por falta de informação nos tíquetes, por falha na coordenação entre desenvolvedores e por falta de conhecimento sobre o processo de desenvolvimento.

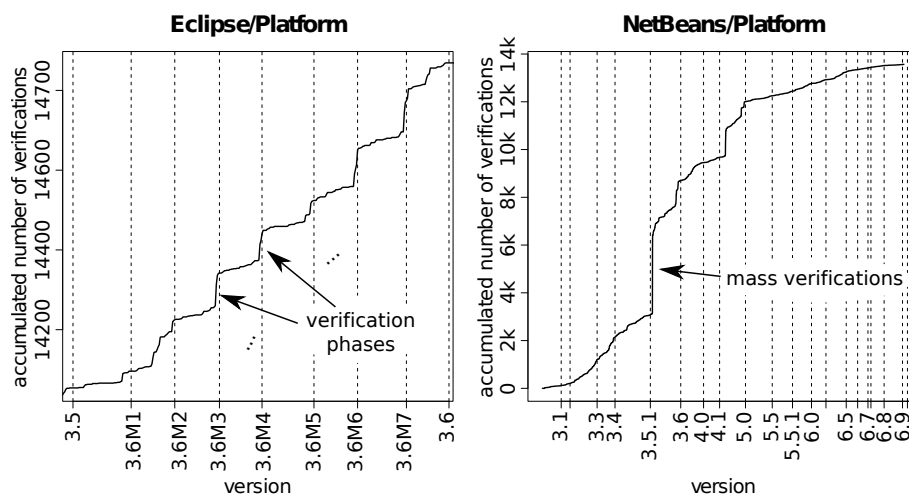


Figura 5. Número acumulado de verificações ao longo do tempo.

5.2.4. Caracterização do Processo de Verificação

Com a finalidade de compreender melhor o processo de verificação nos projetos estudados e de avaliar a utilidade dos dados de sistemas de acompanhamento para auxiliar essa compreensão, foi realizado um estudo exploratório com os projetos Eclipse/Platform, Eclipse/EMF, NetBeans/Platform e NetBeans/VersionControl. O estudo foi apresentado no MSR 2012 (9th Working Conference on Mining Software Repositories) [Souza and Chavez 2012]. O artigo está anexado ao final deste documento.

O estudo foi guiado por três questões de pesquisa, que compõem o primeiro objetivo específico deste trabalho (Seção 3.2):

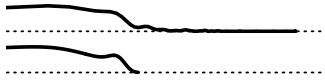
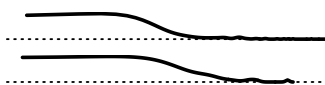
- *Quando* os tíquetes são verificados? Logo após a correção ou em uma fase de verificação?
- *Quem* verifica os tíquetes? Existe uma equipe de qualidade que lida com verificação?
- *Como* é feita a verificação? São realizadas revisões, testes e outras técnicas?

Os resultados são resumidos a seguir.

Quando os tíquetes são verificados? Foram analisadas todas as verificações (i.e., mudanças de *status* para VERIFIED) durante toda a história do projeto. Então foi plotado o número acumulado de verificações relatadas para cada dia, como mostrado na Figura 5. A curva é crescente, com trechos de maior inclinação indicando períodos de verificação. Também foram marcados no gráfico as datas de lançamento de versões. No projeto Eclipse/Platform, nota-se uma intensificação de verificação precedendo imediatamente as datas de lançamento, o que caracteriza esses períodos como fases de verificação. Nos demais projetos, o ritmo de verificação é mais ou menos constante ao longo do tempo.

Verificações em massa. Ao analisar o gráfico do NetBeans/Platform (Figura 5), nota-se uma elevação acentuada próxima ao lançamento da versão 3.5.1. Após uma inspeção detalhada, percebe-se que essa elevação representa mais de 2 mil tíquetes que foram marcados como verificados em menos de 5 horas por apenas um desenvolvedor. A maioria dessas verificações compartilha o mesmo comentário de verificação. A explicação,

Tabela 4. Equipe de qualidade (QA) inferida a partir dos relatórios de defeitos.

Projeto	Tam. do time de QA	% das verificações pelo time de QA	Distribuição da razão (defeitos verificados / defeitos corrigidos) por desenvolvedor
Eclipse			
Platform	4 (2.4%)	1.1%	
EMF	0 (0.0%)	0.0%	
NetBeans			
Platform	25 (18.8%)	80.1%	
V.Control	5 (20.8%)	93.2%	

0.1 1 10 100

Tabela 5. Técnicas de verificação inferidas a partir dos dados.

Projeto	Teste	Inspeção	Ad Hoc
Eclipse/Platform	250 (1.1%)	511 (2.2%)	67 (0.3%)
Eclipse/EMF	21 (1.6%)	1 (0.1%)	0
NetBeans/Platform	88 (0.8%)	5 (0.0%)	0
NetBeans/VersionControl	4 (0.1%)	1 (0.0%)	0

confirmada por outros comentários, é que essas marcações representam uma edição em massa dos tíquetes, para fazer uma limpeza nos dados. Depois dessa descoberta, todas as análises foram refeitas removendo-se verificações em massa.

Quem verifica os tíquetes? Considerou-se que um desenvolvedor é especialista em verificação se ele realiza pelo menos dez vezes mais verificações do que correções. A partir daí, foi considerado que o projeto possui uma equipe de qualidade se os desenvolvedores especialistas em verificação participam de pelo menos metade das verificações do projeto. Percebe-se, pela Tabela 4, que nos subprojetos do NetBeans existe uma equipe de qualidade, composta por cerca de 20% dos desenvolvedores, que é responsável por mais de 80% das verificações. Tal padrão não foi encontrado nos subprojetos do Eclipse.

Como é feita a verificação? Foram analisados os comentários que os desenvolvedores escrevem quando marcam tíquetes como VERIFIED ou REOPENED. Os comentários foram classificados de acordo com a técnica que o desenvolvedor declarou ter usado: testes de unidade/testes automatizados, inspeção/revisão de código ou testes *ad hoc* (isto é, executar o programa de forma não sistemática em busca de falhas). A classificação foi realizada por meio de expressões regulares. A frequência de cada técnica é apresentada na Tabela 5. Infelizmente, no melhor dos casos, foi possível classificar as técnicas em menos de 4% dos comentários. Em uma inspeção informal dos comentários, concluiu-se que a maioria dos comentários não dá pistas sobre a técnica utilizada. Outro aspecto a se considerar é que busca por expressões regulares pode não ser o método mais adequado para obter a informação.

5.3. Influência do Processo de Verificação na Reabertura de Defeitos

Ao final do estudo de caracterização, foram levantadas hipóteses sobre como as características do processo de verificação (quando, quem e como) afetam a reabertura de defeitos:

- nos projetos que possuem uma equipe de qualidade, defeitos cuja correção é verificada por membros da equipe de qualidade têm menos chance de serem reabertos depois de marcados como VERIFIED; a justificativa é que tais pessoas são treinadas para achar defeitos;
- nos projetos que possuem uma fase de verificação, defeitos cuja correção é verificada dentro de uma fase de verificação têm menos chance de serem reabertos; a justificativa é que a verificação requer uma mentalidade diferente da produção de código-fonte, e durante a fase de verificação os desenvolvedores podem se dedicar à verificação, com pouca interferência de atividades de desenvolvimento;
- defeitos verificados por revisão de código têm menos chance de serem reabertos; a justificativa é que testes *ad hoc* são realizados frequentemente, e a revisão de código é uma forma complementar de verificação.

As hipóteses foram estudadas de maneira preliminar, contribuindo para o avanço do terceiro objetivo específico deste trabalho (Seção 3.4). Foram analisados os dados dos projetos Eclipse/Platform, Eclipse/EMF, NetBeans/Platform e NetBeans/VersionControl. Foi usado o teste exato de Fisher para avaliar a dependência entre a reabertura e as diversas características de verificação (verificação por equipe de qualidade ou por desenvolvedor, verificação dentro ou fora da fase de verificação etc.). Os resultados a seguir são preliminares e ainda não foram publicados:

- *Equipe de qualidade vs. reabertura.* Não foi encontrada evidência de que correções realizadas pela equipe de qualidade tenham menos chance de serem reabertas.
- *Fase de verificação vs. reabertura.* Foram encontradas, nos projetos Eclipse/Platform e NetBeans/Platform, evidências estatísticas significativas de que defeitos verificados durante a fase de verificação têm menos chance de serem reabertos (1,29 vezes menos chance no Eclipse/Platform, $p = 0,015$, e 1,75 menos chance no NetBeans/Platform, $p = 0,001$). Nos outros dois projetos não foi encontrada evidência significativa.
- *Revisão de código vs. reabertura.* Foi estudado somente o projeto Eclipse/Platform, no qual foi possível identificar a técnica de verificação empregada em cerca de 4% dos casos. As evidências apontam que verificações que incluem revisão de código têm 4 vezes menos chance de serem reabertos ($p < 0,001$).

Espera-se refinar o estudo a partir da caracterização da reabertura de defeitos (segundo objetivo específico, Seção 3.3), conforme detalhamento apresentado na Seção 3.4. Após a caracterização, espera-se identificar características de defeitos, equipes de desenvolvimento, ciclo de vida do software etc., que podem ter influenciado as análises preliminares e que poderão ser, então, isoladas. Pode-se, por exemplo, identificar que as análises preliminares foram influenciadas pela severidade dos defeitos, e então realizar uma nova análise, controlando a variável severidade.

5.3.1. Princípio dos Quatro Olhos e Reabertura

Outro estudo inferencial já realizado se refere ao princípio dos quatro olhos [Rubin et al. 2007], segundo o qual desenvolvedores que escrevem um certo código-fonte não devem

participar de sua verificação; em vez disso, o código deve ser verificado por outro desenvolvedor. Esse princípio está explícito nas recomendações para desenvolvedores do Eclipse¹⁹. A justificativa é que quem escreve uma correção está muito próximo do código-fonte e pode deixar passar casos extremos.

Espera-se, portanto, que defeitos nos quais a mesma pessoa escreva a correção e realize a verificação tenham uma tendência à reabertura, uma vez que nessa situação acredita-se que é maior a chance de casos extremos serem ignorados durante a verificação. Para avaliar essa hipótese, foram estudados defeitos reabertos e não-reabertos de subprojetos do Eclipse e NetBeans. Para cada defeito, determinou-se se o defeito foi tratado de acordo com o princípio dos quatro olhos ou se ele foi verificado pela pessoa que o corrigiu.

Foram analisados todos os subprojetos do Eclipse e do NetBeans com pelo menos 50 defeitos reabertos. A seguir, foi aplicado o teste exato de Fisher para avaliar a dependência entre as duas variáveis binárias do estudo: se o defeito foi tratado segundo o princípio dos quatro olhos, e se o defeito foi reaberto. Não foi encontrada evidência estatística suficiente para supor que defeitos tratados conforme o princípio dos quatro olhos têm menor chance de ser reabertos.

Uma possível explicação para o resultado é que os desenvolvedores verificam suas próprias correções quando elas são consideradas fáceis, e delegam apenas a verificação de correções mais difíceis. Essa hipótese não foi testada porque não há dados sobre dificuldade de uma correção nos sistemas de acompanhamento de defeitos analisados.

6. Considerações Finais

Este trabalho se propõe a estudar o fenômeno da reabertura de defeitos corrigidos: suas características, o custo associado e meios de diminuir sua ocorrência através da verificação de software. O tema de pesquisa é recente, e por isso há lacunas no conhecimento, sobretudo quanto aos seus impactos e prevenção.

A abordagem de pesquisa se baseia na mineração sistemas de acompanhamento de defeitos. Alguns resultados preliminares já foram obtidos, como a associação entre fase de verificação e a diminuição do número de reaberturas.

A principal limitação deste trabalho é que a validade de suas conclusões depende da qualidade dos dados obtidos dos sistemas de acompanhamento de defeitos. Os dados podem ser imprecisos e conter ruídos. Para mitigar essa ameaça, análises quantitativas e qualitativas são intercaladas, de forma a detectar o quanto antes oportunidades de limpeza nos dados.

Outra consideração importante é a validade de construção, isto é, até que ponto as métricas utilizadas realmente representam o fenômeno que se deseja estudar. Em especial, o tempo para correção (Seção 2.4.2) pode não representar bem o esforço ou o custo de desenvolvimento da correção, uma vez que o tempo é medido a partir da abertura do relatório, e não a partir do início do desenvolvimento da solução. Pode-se argumentar, no entanto, que essa limitação da métrica se aplica igualmente a todos os relatórios de defeitos. Desta forma, é possível mitigar a ameaça ao se realizar apenas análises comparativas com a métrica tempo para correção.

¹⁹http://wiki.eclipse.org/Development_Resources/HOWTO/Bugzilla_Use

Espera-se neste trabalho determinar empiricamente que práticas, sobretudo ligadas à verificação de software, podem ser usadas para diminuir a quantidade de defeitos reabertos durante o desenvolvimento de software. Esse conhecimento, se aplicado, pode reduzir o custo de desenvolvimento e aumentar a qualidade do software produzido.

Referências

- [Almossawi 2012] Almossawi, A. (2012). Investigating the architectural drivers of defects in open-source software systems: an empirical study of defects and reopened defects in gnome.
- [Anbalagan and Vouk 2009] Anbalagan, P. and Vouk, M. (2009). On predicting the time taken to correct bug reports in open source projects. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 523–526.
- [Antoniol et al. 2008] Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., and Guéhéneuc, Y.-G. (2008). Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds, CASCON '08*, pages 23:304–23:318, New York, NY, USA. ACM.
- [Aranda and Venolia 2009] Aranda, J. and Venolia, G. (2009). The secret life of bugs: Going past the errors and omissions in software repositories. In *Proceedings of the 31st International Conference on Software Engineering, ICSE '09*, pages 298–308, Washington, DC, USA. IEEE Computer Society.
- [Arthur et al. 1999] Arthur, J. D., Gröner, M. K., Hayhurst, K. J., and Michael Holloway, C. (1999). Evaluating the effectiveness of independent verification and validation. *Computer*, 32(10):79–83.
- [Ayari et al. 2007] Ayari, K., Meshkinfam, P., Antoniol, G., and Di Penta, M. (2007). Threats on building models from cvs and bugzilla repositories: the mozilla case study. In *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research, CASCON '07*, pages 215–228, New York, NY, USA. ACM.
- [Baker and Eick 1994] Baker, M. J. and Eick, S. G. (1994). Visualizing software systems. In *Proceedings of the 16th international conference on Software engineering, ICSE '94*, pages 59–67, Los Alamitos, CA, USA. IEEE Computer Society Press.
- [Basili et al. 1996] Basili, V. R., Briand, L. C., and Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761.
- [Bhattacharya and Neamtiu 2011] Bhattacharya, P. and Neamtiu, I. (2011). Bug-fix time prediction models: can we do better? In van Deursen, A., Xie, T., and Zimmermann, T., editors, *MSR*, pages 207–210. IEEE.
- [Bird et al. 2009] Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., and Devanbu, P. (2009). Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC/FSE '09*, pages 121–130, New York, NY, USA. ACM.

- [Bird et al. 2006] Bird, C., Gourley, A., Devanbu, P., Gertz, M., and Swaminathan, A. (2006). Mining email social networks. In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 137–143, New York, NY, USA. ACM.
- [Boehm et al. 1976] Boehm, B. W., Brown, J. R., and Lipow, M. (1976). Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 592–605, Los Alamitos, CA, USA. IEEE Computer Society Press.
- [Bougie et al. 2010] Bougie, G., Treude, C., German, D., and Storey, M.-A. (2010). A comparative exploration of freebsd bug lifetimes. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 106–109.
- [Caglayan et al. 2012] Caglayan, B., Misirli, A. T., Miranskyy, A., Turhan, B., and Bener, A. (2012). Factors characterizing reopened issues: a case study. In *Proceedings of the 8th International Conference on Predictive Models in Software Engineering*, PROMISE '12, pages 1–10, New York, NY, USA. ACM.
- [Callahan et al. 1998] Callahan, J. R., Khatsuriya, R., and Hefner, R. (1998). Empirical evaluation of the effectiveness of independent verification and validation (iv&v) through the analysis of automated issue reports on large-scale software projects.
- [Canfora et al. 2011] Canfora, G., Ceccarelli, M., Cerulo, L., and Di Penta, M. (2011). How long does a bug survive? an empirical study. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, WCRE '11, pages 191–200, Washington, DC, USA. IEEE Computer Society.
- [Cavano and McCall 1978] Cavano, J. P. and McCall, J. A. (1978). A framework for the measurement of software quality. In *Proceedings of the software quality assurance workshop on Functional and performance issues*, pages 133–139, New York, NY, USA. ACM.
- [Chrissis et al. 2003] Chrissis, M. B., Konrad, M., and Shrum, S. (2003). *CMMI Guidelines for Process Integration and Product Improvement*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- [D'Ambros et al. 2010] D'Ambros, M., Lanza, M., and Robbes, R. (2010). An extensive comparison of bug prediction approaches. In Whitehead, J. and Zimmermann, T., editors, *MSR*, pages 31–41. IEEE.
- [Duc Anh et al. 2011] Duc Anh, N., Cruzes, D. S., Conradi, R., and Ayala, C. (2011). Empirical validation of human factors in predicting issue lead time in open source projects. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, Promise '11, pages 13:1–13:10, New York, NY, USA. ACM.
- [Eaddy et al. 2008] Eaddy, M., Zimmermann, T., Sherwood, K. D., Garg, V., Murphy, G. C., Nagappan, N., and Aho, A. V. (2008). Do crosscutting concerns cause defects? *IEEE Trans. Softw. Eng.*, 34(4):497–515.
- [Fischer et al. 2003] Fischer, M., Pinzger, M., and Gall, H. (2003). Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, pages 23–, Washington, DC, USA. IEEE Computer Society.

- [Giger et al. 2010] Giger, E., Pinzger, M., and Gall, H. (2010). Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 52–56, New York, NY, USA. ACM.
- [Grady 1992] Grady, R. B. (1992). *Practical software metrics for project management and process improvement*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
- [Hassan 2008] Hassan, A. E. (2008). The road ahead for mining software repositories. *Frontiers of Software Maintenance*.
- [Herzig et al. 2012] Herzig, K., Just, S., and Zeller, A. (2012). It's not a bug, it's a feature: How misclassification impacts bug prediction. Technical report, Universität des Saarlandes, Saarbrücken, Germany.
- [Hooimeijer and Weimer 2007] Hooimeijer, P. and Weimer, W. (2007). Modeling bug report quality. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 34–43, New York, NY, USA. ACM.
- [IEEE Computer Society 2004] IEEE Computer Society (2004). *Software Engineering Body of Knowledge (SWEBOOK)*. Angela Burgess, EUA.
- [ISO/IEC 2001] ISO/IEC (2001). *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC.
- [Jacobson et al. 2007] Jacobson, I., Ng, P. W., and Spence, I. (2007). Enough of processes - lets do practices. *Journal of Object Technology*, 6(6):41–66.
- [Jongyindee et al. 2011] Jongyindee, A., Ohira, M., Ihara, A., and Matsumoto, K.-i. (2011). Good or bad committers? a case study of committers' cautiousness and the consequences on the bug fixing process in the eclipse project. In *Proceedings of the 2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement*, IWSM-MENSURA '11, pages 116–125, Washington, DC, USA. IEEE Computer Society.
- [Kagdi et al. 2007] Kagdi, H., Collard, M. L., and Maletic, J. I. (2007). A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.*, 19:77–131.
- [Kan 2002] Kan, S. H. (2002). *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.
- [Kim and Whitehead 2006] Kim, S. and Whitehead, Jr., E. J. (2006). How long did it take to fix bugs? In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 173–174, New York, NY, USA. ACM.
- [Kim et al. 2007] Kim, S., Zimmermann, T., Whitehead Jr., E. J., and Zeller, A. (2007). Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 489–498, Washington, DC, USA. IEEE Computer Society.
- [Marcus et al. 2008] Marcus, A., Poshyanyk, D., and Ferenc, R. (2008). Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Trans. Softw. Eng.*, 34(2):287–300.
- [McNamee 2005] McNamee, R. (2005). Regression modelling and other methods to control confounding. *Occupational and environmental medicine*, 62(7):500–506.

- [Moser et al. 2008] Moser, R., Pedrycz, W., and Succi, G. (2008). A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 181–190, New York, NY, USA. ACM.
- [Nagappan et al. 2010] Nagappan, N., Zeller, A., Zimmermann, T., Herzig, K., and Murphy, B. (2010). Change bursts as defect predictors. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, ISSRE '10, pages 309–318, Washington, DC, USA. IEEE Computer Society.
- [Nguyen et al. 2010] Nguyen, T. H. D., Adams, B., and Hassan, A. E. (2010). A case study of bias in bug-fix datasets. In Antoniol, G., Pinzger, M., and Chikofsky, E. J., editors, *WCRE*, pages 259–268. IEEE Computer Society.
- [Ohira et al. 2012] Ohira, M., Hassan, A., Osawa, N., and Matsumoto, K. (2012). The impact of bug management patterns on bug fixing: A case study of eclipse projects. In *Proceedings of 28th IEEE International Conference on Software Maintenance (ICSM2012)*.
- [Panjer 2007] Panjer, L. D. (2007). Predicting eclipse bug lifetimes. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 29–, Washington, DC, USA. IEEE Computer Society.
- [Park et al. 2012] Park, J., Kim, M., Ray, B., and Bae, D.-H. (2012). An empirical study of supplementary bug fixes. In Lanza, M., Penta, M. D., and Xi, T., editors, *MSR*, pages 40–49. IEEE.
- [Patton 2005] Patton, R. (2005). *Software Testing (2nd Edition)*. Sams, Indianapolis, IN, USA.
- [Rubin et al. 2007] Rubin, V., Günther, C. W., Van Der Aalst, W. M. P., Kindler, E., Van Dongen, B. F., and Schäfer, W. (2007). Process mining framework for software processes. In *Proceedings of the 2007 international conference on Software process*, ICSP'07, pages 169–181, Berlin, Heidelberg. Springer-Verlag.
- [Rutar et al. 2004] Rutar, N., Almazan, C. B., and Foster, J. S. (2004). A comparison of bug finding tools for java. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, ISSRE '04, pages 245–256, Washington, DC, USA. IEEE Computer Society.
- [Shihab et al. 2012] Shihab, E., Ihara, A., Kamei, Y., Ibrahim, W., Ohira, M., Adams, B., Hassan, A., and Matsumoto, K.-i. (2012). Studying re-opened bugs in open source software. *Empirical Software Engineering*, pages 1–38. 10.1007/s10664-012-9228-6.
- [Shihab et al. 2010] Shihab, E., Ihara, A., Kamei, Y., Ibrahim, W. M., Ohira, M., Adams, B., Hassan, A. E., and Matsumoto, K.-i. (2010). Predicting re-opened bugs: A case study on the eclipse project. In *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, WCRE '10, pages 249–258, Washington, DC, USA. IEEE Computer Society.
- [Slaughter et al. 1998] Slaughter, S. A., Harter, D. E., and Krishnan, M. S. (1998). Evaluating the cost of software quality. *Commun. ACM*, 41(8):67–73.

- [Śliwerski et al. 2005] Śliwerski, J., Zimmermann, T., and Zeller, A. (2005). When do changes induce fixes? In *Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, pages 1–5, New York, NY, USA. ACM.
- [Sommerville 2006] Sommerville, I. (2006). *Software Engineering*. Addison-Wesley.
- [Souza and Chavez 2012] Souza, R. and Chavez, C. (2012). Characterizing verification of bug fixes in two open source IDEs. In Lanza, M., Pent, M. D., and Xi, T., editors, *MSR*, pages 70–73. IEEE.
- [Wagner 2005] Wagner, S. (2005). Towards software quality economics for defect-detection techniques. In *Proceedings of the 29th Annual IEEE/NASA on Software Engineering Workshop*, SEW '05, pages 265–274, Washington, DC, USA. IEEE Computer Society.
- [Wagner 2006] Wagner, S. (2006). A model and sensitivity analysis of the quality economics of defect-detection techniques. In *Proceedings of the 2006 international symposium on Software testing and analysis*, ISSTA '06, pages 73–84, New York, NY, USA. ACM.
- [Wang et al. 2011] Wang, X. O., Baik, E., and Devanbu, P. T. (2011). Operating system compatibility analysis of eclipse and netbeans based on bug data. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 230–233, New York, NY, USA. ACM.
- [Weiss et al. 2007] Weiss, C., Premraj, R., Zimmermann, T., and Zeller, A. (2007). How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 1–, Washington, DC, USA. IEEE Computer Society.
- [Zeng and Rine 2004] Zeng, H. and Rine, D. (2004). Estimation of software defects fix effort using neural networks. In *Proceedings of the 28th Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts - Volume 02*, COMPSAC '04, pages 20–21, Washington, DC, USA. IEEE Computer Society.
- [Zhang et al. 2012] Zhang, F., Khomh, F., Zou, Y., and Hassan, A. (2012). An empirical study on factors impacting bug fixing time. In *Proc. of the 19th Working Conference on Reverse Engineering (WCRE)*.
- [Zimmermann et al. 2012] Zimmermann, T., Nagappan, N., Guo, P. J., and Murphy, B. (2012). Characterizing and predicting which bugs get reopened. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1074–1083, Piscataway, NJ, USA. IEEE Press.

Characterizing Verification of Bug Fixes in Two Open Source IDEs

Rodrigo Souza and Christina Chavez
Software Engineering Labs
Department of Computer Science - IM
Universidade Federal da Bahia (UFBA), Brazil
{rodrigo.flach}@dcc.ufba.br

Abstract—Data from bug repositories have been used to enable inquiries about software product and process quality. Unfortunately, such repositories often contain inaccurate, inconsistent, or missing data, which can originate misleading results. In this paper, we investigate how well data from bug repositories support the discovery of details about the software verification process in two open source projects, Eclipse and NetBeans. We have been able to identify quality assurance teams in NetBeans and to detect a well-defined verification phase in Eclipse. A major challenge, however, was to identify the verification techniques used in the projects. Moreover, we found cases in which a large batch of bug fixes is simultaneously reported to be verified, although no software verification was actually done. Such mass verifications, if not acknowledged, threatens analyses that rely on information about software verification reported on bug repositories. Therefore, we recommend that the exploratory analyses presented in this paper precede inferences based on reported verifications.

Keywords—mining software repositories; bug tracking systems; software verification; empirical study

I. INTRODUCTION

Bug repositories have for a long time been used in software projects to support coordination among stakeholders. They record discussion and progress of software evolution activities, such as bug fixing and software verification. Hence, bug repositories are an opportunity for researchers who intend to investigate issues related to the quality of both the product and the process of a software development team.

However, mining bug repositories has its own risks. Previous research has identified problems of missing data (e.g., rationale, traceability links between reported bug fixes and source code changes) [1], inaccurate data (e.g., misclassification of bugs) [2], and biased data [3].

In previous research [4], we tried to assess the impact of independent verification of bug fixes on software quality, by mining data from bug repositories. We relied on reported verifications tasks, as recorded in bug reports, and interpreted the recorded data according to the documentation for the specific bug tracking system used. As the partial results suggested that verification has no impact on software quality, we questioned the accuracy of the data about verification of bug fixes, and thus decided to investigate how verification is actually performed and reported on the projects that were analyzed, Eclipse and NetBeans.

Hence, in this paper, we investigate the following exploratory research questions regarding the software verification process in Eclipse and NetBeans:

- **When** is the verification performed: is it performed just after the fix, or is there a verification phase?
- **Who** performs the verification: is there a QA (quality assurance) team?
- **How** is the verification performed: are there performed ad hoc tests, automated tests, code inspection?

The next section contains some background information about bug tracking systems and software verification. In Section III, the data and methods used to investigate the research questions are presented. Then, in Section IV, the results are exposed and discussed. Finally, Section V presents some concluding remarks.

II. BACKGROUND

Bug tracking systems allow users and developers of a software project to manage a list of bugs for the project, along with information such as steps to reproduce the bug and the operating system used. Developers choose bugs to fix and report on the progress of the bug fixing activities, ask for clarification, discuss causes for the bug etc.

In this research, we focus on Bugzilla, an open source bug tracking system used by software projects such as Eclipse, Mozilla, Linux Kernel, NetBeans, Apache, and companies such as NASA and Facebook¹. The general concepts from Bugzilla should apply to most other bug tracking systems.

One important feature of a bug that is recorded on bug tracking systems is its status. The status records the progress of the bug fixing activity. Figure 1 shows each status that can be recorded in Bugzilla, along with typical transitions between status values, i.e., the workflow.

In simple cases, a bug is created and receive the status UNCONFIRMED (when created by a regular user) or NEW (when created by a developer). Next, it is ASSIGNED to a developer, and then it is RESOLVED, possibly by fixing it with a patch on the source code. The solution is then VERIFIED by someone in the quality assurance team, if it is adequate, or otherwise it is REOPENED. When a version of the software is released, all VERIFIED bugs are CLOSED.

¹Complete list available at <http://www.bugzilla.org/installation-list/>.

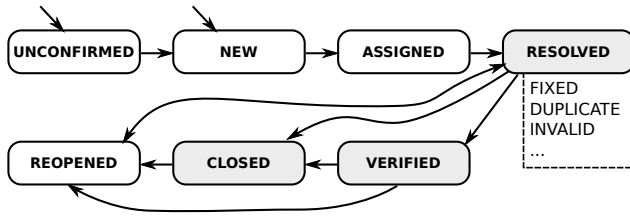


Figure 1. Simplified workflow for Bugzilla. Adapted from <http://www.bugzilla.org/docs/2.18/html/lifecycle.html>.

Bugzilla documentation states that, when a bug is VERIFIED, it means that “QA [quality assurance team] has looked at the bug and the resolution and agrees that the appropriate resolution has been taken”². It does not specify how developers should look at the resolution (e.g., by looking at the code, or by running the patched software).

Software verification techniques are classified in static and dynamic [5]. Static techniques include *source code inspection*, automated *static analysis*, and *formal verification*. Dynamic techniques, or testing, involve executing the software system under certain conditions and comparing its actual behavior with the intended behavior. Testing can be done in an improvised way (*ad hoc testing*), or it can be structured as a list of test cases, leading to *automated testing*.

III. METHOD

In order to answer the research questions—when and how bug fixes are verified, and who verifies them—, a three-part method was used:

- 1) **Data extraction:** we have obtained publicly available raw data from the Bugzilla repositories of two integrated development environments, Eclipse and NetBeans.
- 2) **Data sampling:** for each project, two representative subprojects were chosen for analysis.
- 3) **Data analysis:** for each research question, a distinct analysis was required, as will be further described.

The experimental package is available at <https://sites.google.com/site/rodrigors2/msr2012>

A. Data Extraction

In order to perform the desired analyses, we needed access to the data recorded by Bugzilla for a specific project, including status changes and comments. We have found such data for two projects—Eclipse and NetBeans—from the domain of integrated development environments. The data was made available as part of the Mining Software Repositories 2011 Challenge³ in the form of MySQL database dumps. The files contain all data from the respective databases, except for developer profiles, omitted for privacy reasons.

²<https://landfill.bugzilla.org/bugzilla-3.6-branch/page.cgi?id=fields.html>

³<http://2011.msrfconf.org/msr-challenge.html>

Eclipse development began in late 1998 with IBM⁴. It was licensed as open source in November, 2001. The available data set contains 316,911 bug reports for its 155 subprojects, from October, 2001 to June, 2010.

NetBeans⁵ started as a student project in 1996. It was then bought by Sun Microsystems in October, 1999, and open sourced in March, 2000. The data set contains 185,578 bug reports for its 39 subprojects, from June, 1998 to June, 2010.

B. Data Sampling

Four subprojects were chosen for further analysis: Eclipse/Platform, Eclipse/EMF, NetBeans/Platform, and NetBeans/VersionControl. The Platform subprojects are the main subprojects for the respective IDEs, so they are both important and representative of each projects’ philosophy.

The other two subprojects were chosen at random, restricted to subprojects in which the proportion of verified bugs was greater than the proportion observed in the respective Platform subprojects. The reason is to avoid selecting projects in which bugs are seldom marked as VERIFIED. The following proportions of VERIFIED bugs per project were observed: Eclipse/Platform: 16.0%; Eclipse/EMF: 48.4%; NetBeans/Platform: 21.4%; NetBeans/VersionControl: 29.7%.

C. Analysis: When Are Bugs Verified?

In order to determine if there is a well-defined verification phase for the subprojects, we have selected all reported verifications (i.e., status changes to VERIFIED) over the lifetime of each subproject. Then, we have plotted, for each day in the interval, the accumulated number of verifications reported since the first day available in the data. The curve is monotonically increasing, with steeper ascents representing periods of intense verification activity.

Also, we have obtained the release dates for multiple versions of Eclipse and NetBeans. The information was obtained from the respective websites. In cases in which older information was not available, archived versions of the web pages were accessed via the website www.archive.org.

If a subproject presents a well-defined verification phase, it is expected that the verification activity is more intense a few days before a release. Such pattern can be identified by visual inspection of the graph, by looking for steeper ascents in the verification curve preceding the release dates.

D. Analysis: Who Verifies Bugs?

In order to determine whether there is a team dedicated to quality assurance (QA), we have counted how many times each developer has marked a bug as FIXED or VERIFIED. We considered that a developer is part of a QA team if s/he verified at least 10 times (i.e., one order of magnitude) more than s/he fixed bugs. Also, we have computed the proportion

⁴<http://www.ibm.com/developerworks/rational/library/nov05/cernosek/>

⁵<http://netbeans.org/about/history.html>

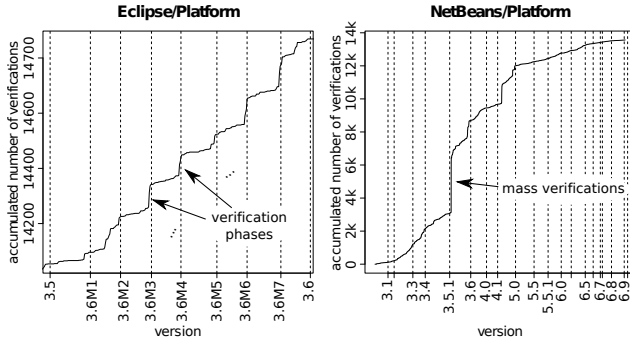


Figure 2. Accumulated number of verifications over time.

of verifications that was performed by the discovered QA team. It is expected that, if the discovered set of developers is actually a QA team, they should be responsible for the majority of the verifications.

E. Analysis: How Are Bugs Verified?

In order to discover the verification techniques used by the subprojects, we have selected the comments written by developers when they mark a bug as `VERIFIED` (meaning that the fix was accepted) or `REOPENED` (meaning that the fix was rejected).

The comments were matched against five regular expressions, each corresponding to one of the following verification techniques: automated testing, source code inspection, ad hoc testing, automated static analysis, and formal testing. The complete regular expressions are available in the experimental package.

It should be noted that regular expressions may not be a reliable alternative to the problem of identifying verification techniques in comments. In future research, more advanced information retrieval techniques should be explored. Nevertheless, regular expressions enable an initial study and help unveil insights about verification techniques in bug reports.

IV. RESULTS AND DISCUSSION

Each of the three analyses was performed on Eclipse/Platform, Eclipse/EMF, NetBeans/Platform, and NetBeans/VersionControl. The results are presented next.

A. When Are Bugs Verified?

Figure 2 shows plots of total accumulated number of verifications over time for Eclipse/Platform (left) and NetBeans/Platform (right). Releases are plotted as dashed vertical lines.

For Eclipse/Platform, the graph shows the period between releases 3.5 and 3.6, including milestone releases every 6 or 7 weeks. It is clear from the graph that the verification activity is intensified in a few days preceding a milestone, represented in the graph by steeper ascents before the vertical lines. This pattern is an indicator of a verification

Table I
DISCOVERED QA TEAM FOR ALL FOUR SUBPROJECTS.

Project	QA team size	% of verifications by QA team	Distribution of ratio (bugs verified / bugs fixed) per developer
Eclipse			
Platform	4 (2.4%)	1.1%	
EMF	0 (0.0%)	0.0%	
NetBeans			
Platform	25 (18.8%)	80.1%	
V.Control	5 (20.8%)	93.2%	

phase. No such pattern was found in the other subprojects by analyzing their graphs (not shown here for brevity).

The graph for NetBeans/Platform (Figure 2, right side) shows the entire project history. Although there are steeper ascents, they are different because they do not precede release dates. Also, at a closer look, they represent thousands of verifications performed in a few minutes by the same developer, with the same comment. The same pattern was found in Eclipse/EMF (not shown).

Of course, no developer can verify so many fixes in so little time. The explanation, supported by the comments, is that such mass verifications represent some cleanup of the bug repository, by marking old bugs as `VERIFIED`—with no verification being actually performed. Researchers should take extra care with mass verifications, as they contain a large amount of bugs and, thus, are likely to bias the results of analyses.

In the next two analyses (who and how), mass verifications were discarded. A verification was considered to be part of a mass verification if the developer who performed it also performed at least other 49 verifications in the same day. Although further research is needed to evaluate such criterion, it was able to identify the most obvious mass verification cases.

After applying the criterion to identify mass verifications, 362 (2.4%) verifications were discarded from Eclipse/Platform, 2348 (72.3%) verifications were discarded from Eclipse/EMF, and 5336 (39.3%) verifications were discarded from NetBeans/Platform. Mass verifications were not identified in NetBeans/VersionControl.

B. Who Verifies Bugs?

Table I presents, for each subproject, the number of developers attributed to a QA team (i.e., developers with verifications per fix ratio greater than the threshold of 10), and the proportion of bug verifications they account for. Other threshold values (2 and 5) were also tried for the ratio, leading to similar results. Mass verifications and developers who have not contributed with fixes and verifications were discarded from the analysis.

In the Eclipse subprojects, there is no evidence of a dedicated QA team. In both NetBeans subprojects, on the

Table II
VERIFICATION TECHNIQUES FOR ALL FOUR SUBPROJECTS.

Project	Testing	Inspection	Ad Hoc
Eclipse/Platform	250 (1.1%)	511 (2.2%)	67 (0.3%)
Eclipse/EMF	21 (1.6%)	1 (0.1%)	0
NetBeans/Platform	88 (0.8%)	5 (0.0%)	0
NetBeans/VersionControl	4 (0.1%)	1 (0.0%)	0

other hand, it is possible to infer the existence of a QA team, composed by approximately 20% of the developers, performing at least 80% of all verifications. Further evidence was found by looking for the substring “QA team” in the comments associated with the status VERIFIED and REOPENED. Comments referring to a QA team were only found in NetBeans subprojects.

C. How Are Bugs Verified?

Table II shows, for each subproject, the number of comments associated with a bug being marked as VERIFIED or REOPENED that refers to a particular verification technique, as matched by the respective regular expressions. Comments associated with mass verifications were discarded.

No references to formal verification were found. Regarding static analysis, only 4 references were found; upon inspection, it was found that only one reference (bug report 15242 for NetBeans/Platform) implies the use of a static analysis tool in the verification process.

In all projects, comments suggest the use of automated testing and code inspection in the verification process, the former technique being more frequently referenced (except in Eclipse/Platform). Evidences of ad hoc testing were found only in Eclipse/Platform, probably due to limitations in the regular expressions.

The regular expressions were able to identify the verification technique only in 3.6% of the comments at best (Eclipse/Platform). Such low proportion can be explained partly by limitations of the regular expression method (which can be addressed in future work) and partly by lack of information in the comments themselves.

By reading comments, we have found that many of them mention any verification technique. Most often, developers just state that the bug fix was verified, sometimes informing the build used to verify the fix.

In Eclipse/Platform, comments show that the developer who fixes a bug often asks someone else to verify the fix, by asking “please verify, [developer name]”. If the bug is reopened, then the fixer and the verifier exchange roles. This behavior illustrates a structured bug fixing/verification process and supports the conclusion that there is no QA team in Eclipse/Platform.

In Eclipse/EMF, we found that marking a bug as VERIFIED does not mean that the bug fix was verified. Instead, it means that the fix was made available in a build of the software that is published in the subproject’s website⁶.

V. CONCLUSION

By analyzing four subprojects (two from Eclipse, two from NetBeans), we have found, using only data from bug repositories, subprojects with and without QA teams, with and without a well-defined verification phase. We also have found weaker evidence of the application of automated testing and source code inspection. Also, there were cases in which marking a bug as VERIFIED did not imply that any kind of software verification was actually performed.

The main threat to this study is related to construct validity, i.e., to which point the operational definitions of concepts such as QA team and verification phase reflect the theoretical concepts. To mitigate this problem, we have looked for additional information in the text of comments in the bug reports. Further evidence can be gathered by interviewing developers and mining source code repositories.

With the knowledge obtained from this exploratory research, we aim to improve and extend our previous work on the impact of independent verification on software quality. We can investigate, for example, whether verification performed by QA team is more effective than verification performed by other developers.

Researchers should be aware that information about verification techniques may not be common in bug repositories, and that reported verification does not always correspond to actual verification (e.g., in the case of mass verifications). Some exploration of the data is important to avoid such pitfalls.

ACKNOWLEDGMENT

This work is supported by FAPESB under grant BOL0119/2010 and by CNPq/INES under grant 573964/2008-4.

REFERENCES

- [1] J. Aranda and G. Venolia, “The secret life of bugs: Going past the errors and omissions in software repositories,” in *Proc. of the 31st Int. Conf. on Soft. Engineering*, 2009, pp. 298–308.
- [2] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, “Is it a bug or an enhancement?: a text-based approach to classify change requests,” in *Proc. of the 2008 Conf. of the Center for Adv. Studies on Collaborative Research*, ser. CASCON ’08. ACM, 2008, pp. 23:304–23:318.
- [3] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, “Fair and balanced?: bias in bug-fix datasets,” in *European Soft. Eng. Conf. and Symposium on the Foundations of Soft. Eng.*, ser. ESEC/FSE ’09. ACM, 2009.
- [4] R. Souza and C. Chavez, “Impact of the four eyes principle on bug reopening,” Universidade Federal da Bahia, Tech. Rep., 2011.
- [5] I. Sommerville, *Software engineering (5th ed.)*. Addison Wesley Longman Publish. Co., Inc., 1995.

⁶See http://wiki.eclipse.org/Modeling_PMC_Meeting,_2007-10-16